A PLIABLE HYBRID ARCHITECTURE FOR CODE ISOLATION

A Dissertation
Presented to
The Academic Faculty

by

Ivan B. Ganev

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
August 2007

A PLIABLE HYBRID ARCHITECTURE FOR CODE ISOLATION

Approved by:

Karsten Schwan, Adviser
College of Computing
Georgia Institute of Technology

Santosh Pande
College of Computing
Georgia Institute of Technology

Mustaque Ahamad
College of Computing
Georgia Institute of Technology

Kiran Panesar
Google Inc.

Greg Eisenhauer
College of Computing
Georgia Institute of Technology

Date Approved: 9 April 2007

For Dana and Ginka.

ACKNOWLEDGEMENTS

For the successful completion of this work, I owe an enormous debt of gratitude to many people who have so generously given me their friendship, encouragement, and time. Perhaps too many to list individually, yet, I would be remiss not to try. Thus, in no particular order, I would like to thank:

- Karsten Schwan, for giving me complete freedom to pursue my research interests and for always being so supportive, understanding, resourceful, and wise. I could not have wished for a better adviser.
- Greg Eisenhauer, for sharing his technical expertise and solid foundation of code.
- Kiran Panesar, Mustaque Ahamad, and Santosh Pande, for their guidance and helpful suggestions.
- Ken Mackenzie, for being a source of true inspiration.
- Josh Fryman, for so many enlightening discussions, ruthless paper revisions, and his contagiously positive attitude.
- Craig Ulmer, for all the jokes and encouragement.
- Richard West, Christian Poellabauer, Himanshu Raj, Sanjay Kumar, Balasubramanian Seshasayee, and countless other fellow Georgia Tech students for their time and friendship.
- My parents, Denka and Boris Ganev, for their constant love and support.
- Last but not least, my lovely wife Ginka, for always being there for me and for putting up with the endless days and nights I spent hacking in the lab, and our beautiful baby Dana, for lighting up our days.

This dissertation truly would not have been possible without you. Thank you!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF LISTINGS

SUMMARY

The unprecedented growth of computing power and communication bandwidth in the last few decades has driven an explosion in the size and complexity of application software. Specifically, it has spurred an almost universal adoption of modular and extensible software designs, from ordinary PC applications, to operating systems kernels, and even to embedded systems. In many cases, however, the ability to extend software systems has come hand in hand with the need to isolate them from untrusted or potentially faulty extensions.

This dissertation will focus on the important problem of code isolation, where existing techniques vary in many and often interrelated dimensions such as granularity, code complexity, invocation latency, dynamism, isolation strategy, permissible extension functionality, and degree of integration with the operating system kernel. Specifically, the implementation of a particular technique imposes restrictions on the properties of extensions. Examples include proof-based techniques that are only applicable to simple extensions of small granularity, hardware-based isolation techniques that typically incur a measurable invocation latency due to hardware re-configuration overhead, and programming language techniques that impose implementation and compiler restrictions.

The goal of this dissertation is to explore the design space of code isolation techniques, identify characteristics of individual approaches, and then argue for and design a *hybrid* approach that combines their advantages while avoiding their drawbacks. The contributions of this thesis will be threefold: (1) a taxonomy of metrics and properties relevant to software code isolation techniques, (2) the design and implementation of a novel *hybrid* architecture for safe kernel extension with pliable characteristics, and (3) an evaluation of the hybrid approach and comparison with homogeneous alternatives.

CHAPTER I

INTRODUCTION

1.1 Background

In the last three decades, the world has enjoyed a tremendous and sustained technological rate of growth in terms of both the available computing and communication speeds and of their penetration in society. Computing power has followed Moore's Law of exponential growth, doubling every 18 months and resulting in an annualized increase of about 60%. Communication bandwidth has also grown exponentially, albeit at a slower annualized rate of approximately 50%, and storage capacities, which have been lagging behind in relative terms, are currently staging a comeback with an annualized growth rate of 100%. All of these trends combined have revolutionized society and sped up almost every aspect of modern life.

The resulting waves of innovation are generating new applications and improving existing ones every day, and the pace is accelerating. To cope with the unrelenting pressure for change, software systems have been evolving away from their static and inflexible origins and towards more modular and adaptable designs. Extensibility has emerged as a property essential for adaptation, and there exists a wide variety of approaches for software extension.

Virtually every type of software system today, from ordinary personal computer (PC) applications, to operating system (OS) kernels, to exotic embedded systems employ or benefit from some form of extensibility technology. Irrespective of their size, purpose, or position in the software hierarchy, applications are making increasing use of plugins, extensions, loadable drivers, or even plain shared libraries. Examples abound, and we briefly mention a few representative ones.

User Software

Popular multimedia applications like XMMS [1], Xine [108], and MPlayer [72] (rich multi-format audio and video players), and others such as the FireFox [71] web browser or the

GIMP [31] image manipulation program have structures that are explicitly designed to allow them to be augmented with or to modify their functionality through new extensions. In the cases of XMMS, Xine, and MPlayer, a large number of *plugins* are available to enable those applications to reproduce a wide variety of media formats, to deal with a rich set of output devices, or even to simply give a custom look to their graphical user interfaces. In the FireFox and GIMP cases, *extensions* are even more versatile and provide a primary mechanism to adapt them to handle new web formats (*e.g.*, Shockwave [3] and Flash animations), to behavioral changes (*e.g.*, pop-up ad blocking or tabbed browsing in FireFox and new image processing algorithms in the GIMP), and even completely new services (*e.g.*, the BugMeNot [23] compulsory web registration bypass service). It is worth mentioning that Emacs [28], one of the most popular text editors, is also the most extensible one. Its wealth of *packages* is so large that Emacs transcends the boundaries of a mere application and has turned into a complete work environment!

System Software

System software is code that is not under user control and is typically vital for the operation of the system. For example, almost any executable running on top of a modern operating system today makes use of *shared objects* (also known as dynamically linked libraries in Microsoft Windows) that help to extend or adapt it to the peculiarities of the particular execution environment. Examples are numerous and span a great range of libraries for any given purpose from versions of OpenGL or DirectX graphics interfaces to the Linux Pluggable Authentication Modules. A crucial advantage of shared objects *vs.* static libraries is that, by acting as physically separate extensions, they afford great flexibility in altering how individual applications interact with the system without having to modify or recompile the actual applications.

OS Kernel

Because of their many benefits, extensions have found a place not only in applications, but also in the underlying operating system kernels. Practically all major OS kernels in use today support some form of extension. Initially spurred by the desire to be able to load

and unload device drivers as needed on the fly, the technology was generalized for arbitrary functionality resulting in the modern *loadable kernel modules* [19] that help to reduce a kernel's memory footprint and enable piecemeal dynamic runtime upgrades.

Pre-Boot Environment

Finally, as the ultimate testament to the usefulness of extensible design, we briefly mention its use in software components even below the level of the OS kernel. The Extensible Firmware Interface (EFI) is a recent Intel initiative to abstract the pre-boot environment away from the details of the underlying BIOS or firmware and turn it into a uniform, device-independent, and extensible environment.

1.2 Motivation

Clearly, extensibility has proved to be a tremendously beneficial technique. Indeed, a well-accepted approach for increasing the adaptability of software systems is to equip them with extension capabilities and use those to modify or increase their functionality at runtime. However, this new found flexibility comes at the price of an increase in the system's fragility or vulnerability because of potentially buggy or even malicious extensions. Exposure to such extension-borne risks is particularly important for privileged software systems like OS kernel, but applies and is still important in unprivileged applications like web browsers or multimedia players. In order to avoid failures, untrusted extensions need to be isolated from the software they augment. This inherent asymmetry of trust leads to the following two requirements to any extension architecture:

1. *Safety*: It must prevent the compromise of the safety of its host software, and
2. *Performance*: Its overheads should be small, lest they overwhelm or render useless the benefits derived through the extensions.

Unfortunately, these requirements are antagonistic. One simple example of their conflict is as follows. Attaining high performance by executing native code in the kernel's address space is possible but would ignore safety issues. Interpreting the untrusted code in a sandbox would provide the desired safety but at the same time it will decrease performance

significantly. Despite these difficulties, OS kernel extensions have proved to be a useful adaptation and specialization model. Perhaps not coincidentally, one of the world's most popular OSs, Microsoft Windows, is also said to be one of the most extensible ones [104].

A large body of research has accumulated over the last decade focusing on ways of safely extending OS kernels to specialize them or to adapt them to individual application needs [12, 22, 102, 16, 29, 77, 51, 91, 103, 30, 13, 61, 84, 85].

Research approaches have employed theoretical, hardware-, or software-based methods for code isolation. Each of these three classes of methods can be used to provide varying levels of safety and performance and has its own general set of properties and characteristics common to its members. We touch briefly on the relative qualities of these three general classes:

1. *Formal Methods* use logic reasoning techniques to prove invariants of extensions and safety properties about the behavior of their code. While their use in restricted problem domains, *e.g.*, the Microsoft driver verifier [8], has been successful, more general approaches, such as Proof-Carrying Code [77] for example, have failed to scale up and have remained practical only for limited code sizes.

2. *Type-Safe Language* approaches like SPIN [12], the Open Kernel Environment [13], or more recently, Cyclone [51] and Microsoft Research's Singularity [44] are based on special properties of languages and/or trusted compilers. The restrictions they impose on development environments, *e.g.*, the Modula-3 language, SPIN's special compiler or Singularity's Sing# language, detract from their practicality and use outside of research lab prototypes.

3. *Software-Based* approaches fall into two subcategories: (1) interpreters and (2) software fault isolation (SFI). Interpreters like the Berkeley Packet Filter [61] trade speed for safety, and performance degradation makes them less fitting for critical infrastructure. SFI [102] rewrites untrusted machine code, inserting runtime protection checks for memory references and jump-targets. Despite achieving performance superior to that of interpreters, SFI approaches still incur a measurable performance

penalty [77, 16] and require intricate and complex binary manipulations.

4. *Hardware-Based* techniques [53, 22, 107, 16] have formed a popular avenue of attack to the problem. Generally, they are able to achieve native performance by configuring the execution environment so that necessary runtime safety checks are carried out by the CPU protection hardware. High performance is attained in terms of avoiding per-instruction overheads, but limitations are imposed by various details of the protection hardware available, *e.g.*, increased latency of invocation compared to a simple procedure call. Additionally, there might be a need to modify certain OS components. However, today's largely mono-cultural hardware world, among other reasons, is mitigating the impact of that latter criticism and making this approach increasingly popular.

This short feature review is by no means comprehensive. Rather, it is only meant to highlight that there are both distinct advantages and distinct drawbacks to each class of techniques, as well as each individual technique in particular. The main point is that no single proposal is capable of satisfying the complete spectrum of features along the multiple dimensions of granularity, code complexity, invocation latency, dynamism, isolation strategy, permissible extension functionality, and degree of integration with the host kernel.

This thesis will focus on the need for an extension architecture that is sufficiently malleable to address the extremes of the medley of assorted requirements as well as combinations of them. We propose a **hybrid** approach to code isolation as we believe it provides a unique opportunity for overcoming the drawbacks of individual techniques.

1.3 Terminology

This section briefly defines the meaning of some terms and notions in the context of this thesis.

OS Extension: An OS extension, henceforth simply termed extension, is the logical unit of functionality that an application deposits into the OS kernel in order to adapt some aspect of it. Extensions are deployed to either modify the behavior or to enhance the

functionality of a kernel service, subsystem, or driver. Examples include, but are not limited to, custom metrics to a thread scheduler, algorithms for page victim selection, networking filter code, *etc.* They typically consists of one or a number of routines.

Routine: Routines are the basic building blocks of extensions. As in regular programs, they serve to impart structure or to encapsulate commonly repeated functionality. Routines can return values like functions or simply serve as procedural containers for code.

Namespace: A namespace refers to the set of symbols that are available to an extension during its runtime. This set consists of the symbols corresponding to its routines, supplied by the extending application, joined with the set of kernel symbols explicitly exported to that extension by the kernel service or subsystem that it adapts. The latter type of symbols need not necessarily refer to internal kernel interfaces directly. Instead, they might refer to interposed wrapper functions that check, limit, or sanitize the kernel-extension interaction and information exchange during runtime.

Latency: Latency is a metric defined as the period of apparent inactivity between the time that a stimulus is presented and the moment that a response occurs. In the context of this dissertation, the latency overhead is an important parameter intrinsic to the various isolation technologies employed to ensure the host kernel's safety from untrusted extensions. Specifically, latency will refer to the amount of wall-clock time elapsing between the execution of the first machine instruction of a kernel to extension invocation and the execution of the first instruction of the extension, or vice versa. Essentially, latency is a measure of how "reactive" different extension isolation technologies are.

Throughput: Throughput is another metric, defined as the rate at which a processor can perform some action expressed as units of work per units of time. In the context of this dissertation, throughput is another important parameter intrinsic to the various isolation technologies employed. Specifically, throughput will refer to the number of times that a null extension can be executed by the host kernel in a single unit of time. Thus, throughput will be used as a measure of the pure cost of some isolation technique.

1.4 Thesis

Modern servers and desktops are general purpose systems, yet they are increasingly required to perform an ever widening number of specialized tasks. Their oftentimes conflicting demands require OS kernels to be adaptable through custom extensions.

This dissertation's thesis is that:

> **A hybrid approach to safe runtime OS kernel adaptation is capable of increased functionality and flexibility thanks to combining the advantages of a variety of individual isolation techniques while avoiding their respective drawbacks. The resulting hybrid kernel extensions are malleable and more general than homogeneous alternatives.**

The goal of the dissertation is to explore the design space of isolation and extension techniques, identify characteristics of individual approaches, and design a *hybrid* architecture that combines their advantages while avoiding their drawbacks.

The contributions of this dissertation are threefold:

1. A taxonomy of metrics and properties relevant to software extensions and an associated classification of existing approaches.

2. Design and implementation of a novel hybrid infrastructure for safe kernel adaptation that enables extensions isolated through a hybrid combination of different techniques.

3. Evaluation and comparison of the proposed hybrid approach to homogeneous alternatives.

1.5 Organization

The remainder of this dissertation is organized as follows:

Chapter 2 introduces a set of isolation requirements, a formal completeness argument, and a classification system based on it.

Chapter 3 presents a novel hybrid code isolation technique based on complementary elements of software- and hardware-fault isolation.

Chapter 4 details the experimental evaluation of the proposed hybrid isolation technique and hybrid extension architecture and compares them against homogeneous alternatives.

Chapter 5 reviews related work on extensible kernels.

Chapter 6 contains some concluding remarks and suggestions for future work.

CHAPTER II

ISOLATION REQUIREMENTS

Attempts to characterize any set of distinct programmatic techniques in general, and code isolation techniques in particular, requires that we compare and contrast descriptions of the alternative techniques' characteristic features. A systematic approach to such a comparison becomes all the more important when one's goal is not only to characterize, but also to compare and exploit their differences as in the hybrid approach presented in this dissertation. To aid in the latter, this chapter is devoted to formulating a complete set of requirements that any code isolation technique must satisfy and then proposing an associated classification.

The invocation of foreign code involves a transfer, and thus at least a temporary loss, of execution control. It also involves the risk of disallowed or undesirable data modification by the foreign code. Informally, it seems natural and self-evident then, that the following characteristics of untrusted code must be controlled:

1. its access to data,
2. its ability to direct the flow of control,
3. its ability to influence the execution environment, and
4. its running time.

We assert that this set of requirements is complete in the sense that it provides for the complete isolation or shielding of a trusted application from an untrusted or foreign code extension. In order to provide a formal foundation for our assertion, we next formulate a completeness argument in the context of a Universal Turing Machine (UTM). This formulation also allows us to simplify the problem by abstracting away many complex implementation details of modern computing architectures.

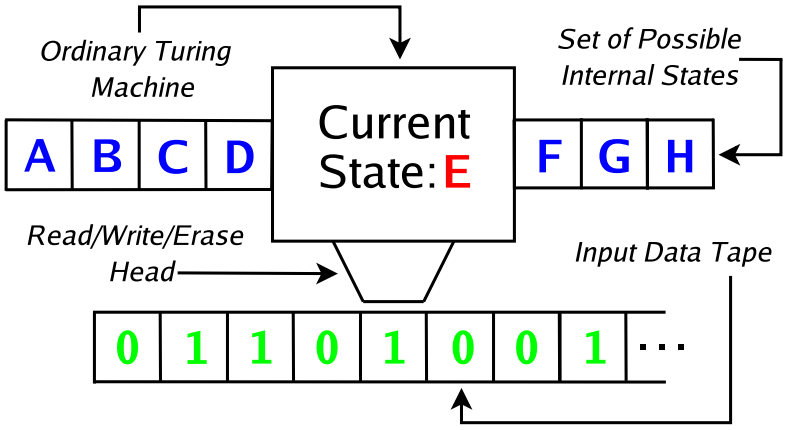

Figure 1: An Ordinary Turing Machine.

2.1 Formal Basis

Turing machines are simple abstract computational devices initially proposed as convenient means of investigating the extent and limitations of what can be computed. They were first described by Alan Turing in 1936 [98] and despite their simplicity, they have been shown to be able to simulate the logic of any computer that can be constructed.

Many possible definitions for what constitutes a Turing machine exist, ranging from the mathematically abstract state automata to the physically feasible, albeit impractical, mechanical automatons [67]. As they have all been shown to be equivalent, we choose to use the simplest single-tape mechanical description of such a machine for our completeness arguments.

A single-tape Ordinary Turing Machine (OTM), such as the one shown in Figure 1, is essentially defined as an automaton that has only a finite number of internal states and an *infinitely* long data tape on which it can read, write, and erase symbols. The tape can move one space at a time in either direction.

While general, Ordinary Turing Machines are static in the sense that each machine is limited to implementing the one algorithm for which its internal states are expressly constructed. In contrast, a Universal Turing Machine can perform any operation of any other Ordinary Turing Machine. This is achieved by means of complex instructions provided on its tape along with input data and in effect make it programmable.

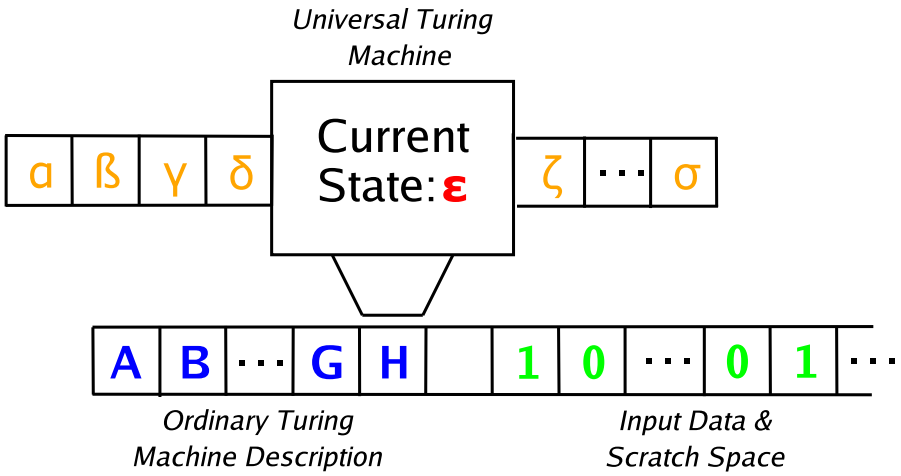

Figure 2: A Universal Turing Machine.

A Universal Turing Machine, like the one depicted in Figure 2, receives a complete symbolic description of the machine it is expected to emulate on one (finite) part of its tape. The rest of the unbounded tape is used for storing input/output data, as well as for scratch space on which the UTM can record the current state of the imitated machine, the tape position it is scanning, *etc.*

As described, Universal Turing Machines possess a number of key characteristics that make them an excellent model in which to study the completeness of our set of requirements for code isolation. Those characteristics are:

1. *Programmable*: since UTMs are programmable, they constitute a direct and intuitive analogue to any modern digital computer.

2. *von Neumann*: because, loosely speaking, the 'code' or program being executed resides on the same tape as the input/output 'data' being operated on, UTMs have an inherently von Neumann architecture closely resembling that of modern digital computers.

3. *Simple*: UTMs have an uncomplicated structure, despite being computationally equivalent to modern machines. This is important, as it allows our arguments to sidestep many nonessential details of actual hardware, such as user/kernel modes, privilege

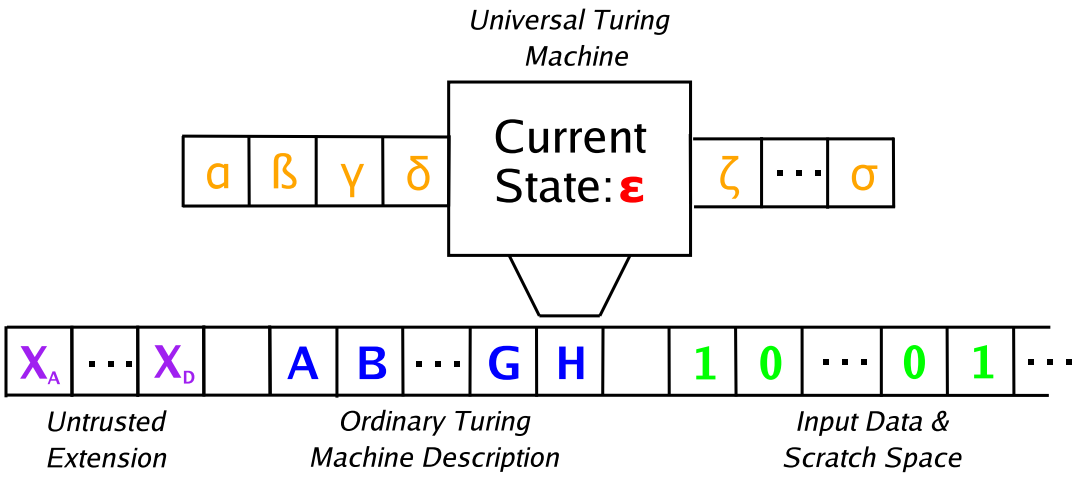

Figure 3: A Universal Turing Machine interpreting an extended Ordinary Turing Machine.

rings, interrupts and exceptions, caches, *etc.*

In the remainder of this chapter, we will consider an Universal Turing Machine that is simulating an Ordinary Turing Machine (the application), the state-description of which has been augmented with additional states (the extension). Our aim will be to draw a direct analogy with a general purpose computer running an application that has been augmented with an extension – a plugin, dynamic library, *etc.* An underlying assumption is that in contrast to the main program, the extension is untrusted and must be isolated and prevented from interfering with the latter.

Figure 3 is a visual representation of the backdrop for the discussion in the rest of this chapter. It features a Universal Turing Machine depicted by a rectangle with an inverted trapezoid read/write head and a finite number of UTM machine states represented with Greek letters comprising its fixed simulation logic. Below it is the UTM's infinite tape split into three regions: two finite 'code' regions and an infinite 'data' region. The former are represented by capital Latin letters with the letters A-G representing the description of the application simulated by the UTM and X_A-X_D representing the additional state description of the untrusted extension. The remainder of the tape extends infinitely to the right and represents both the application's and the extension's input/output data and scratch space.

Albeit simple in structure and with strictly finite state descriptions, Turing machines

can describe arbitrarily complex algorithms thanks to the unlimited amount of scratch space that can be stored on their infinite tapes. The deliberate lack of a limit is a critically important for the expressive power of the machines and must be preserved with respect to both the application and the extension. Thus, it merits a brief mention that the setup described in Figure 3 can accommodate this infinite nature, *e.g.* by assigning positions in the unbounded right portion of the tape to the application and to the extension in an alternating fashion.

The remainder of this chapter concerns the vulnerabilities of machines. Toward this end, we enumerate the individual components of our Universal Turing Machine and systematically analyze their avenues of exposure – the ways in which their functions can be taken advantage of or interfered with by malicious or buggy extensions.

2.2 Attack Types

In the computer security literature, it is widely accepted [95, 81] that a secure system must have all of the following properties:

- **Confidentiality**: A system should only be read accessible to authorized parties,
- **Integrity**: System assets should only be modifiable by authorized parties,
- **Availability**: System assets should be available to authorized parties, and
- **Authenticity**: A system should be able to verify the identity of its users.

The purpose of any computer security system is to ensure that all of these properties are always valid. The types of threats to these properties fall into 4 major categories, as identified by Stallings [95] and Oppliger [81]:

- **Interruption** is an attack on Availability whereby the normal operation of the system is interrupted in some way or is completely denied,
- **Interception** is an attack on Confidentiality whereby an unauthorized party is able to gain access to an asset,

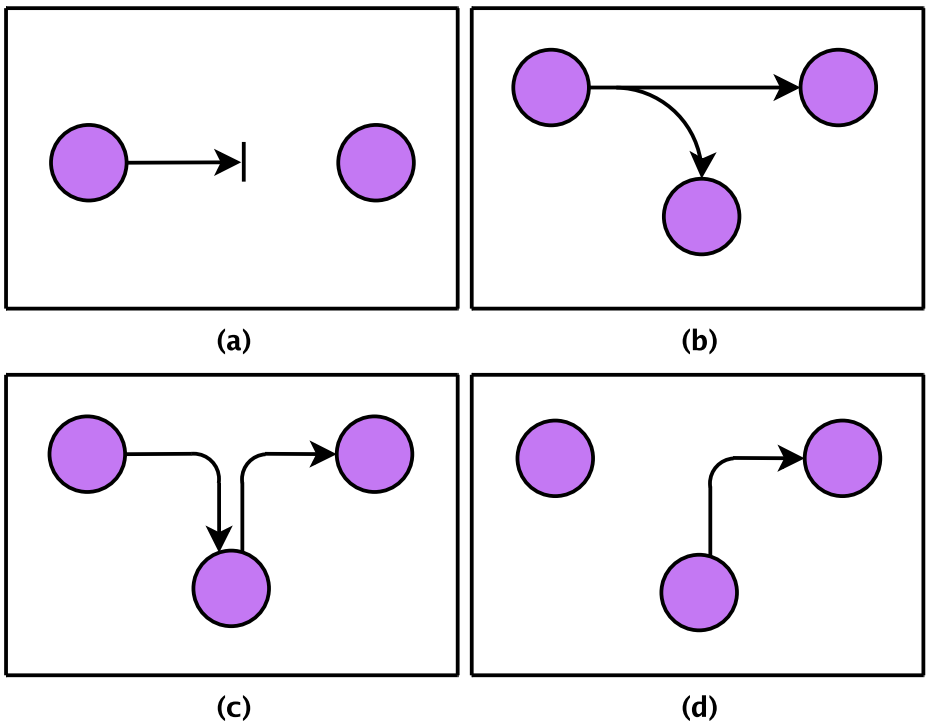

Figure 4: Types of attacks: (a) Interruption, (b) Interception, (c) Modification, (d) Fabrication.

- **Modification** is an attack on Integrity whereby an unauthorized party not only gains access but also tampers with an asset, and
- **Fabrication** is an attack on Authenticity whereby an unauthorized party is able to impersonate an authorized party and/or to insert counterfeit objects into the system.

Figure 4 provides a simple visual representation of the 4 categories of attacks in terms of abstract communicating entities. Bearing this classification in mind, we now focus on the component objects that make up our UTM model. We analyze the types of attacks applicable to each of them, and finally distill a formally complete set of isolation requirements.

2.3 Attack Targets

The simple structure of the Universal Turing Machine in Figure 3 makes it easy to identify an exhaustive list of components. The short list includes the following:

- **State Automaton**: The UTM state automaton consists of the UTM 'execution unit', which holds the current state and realizes its state transitions, as well as the finite set of all possible states – α through σ. The former can be viewed as analogous to a central processing unit and the latter, perhaps, to its microcode.

- **Read/Write Head**: The UTM's read/write head is the inverted trapezoid situated below the state automaton. It provides the UTM with access to both its program and its data. It is analogous to a modern computer's I/O hardware such as I/O-buses, disk controllers, *etc.*

- **Tape**: The UTM tape is the endless strip of symbols sitting immediately below the UTM's read/write head. It corresponds to a computer's temporary and persistent storage systems such as RAM memory, hard disks, CD/DVD drives, *etc.*

In terms of our attack model, we will regard the UTM "hardware", as well as the main application description as benevolent and trustworthy. Hence, we concentrate our attention, first, on identifying the ways in which untrusted extension software can manipulate them to interfere with their proper operation, and second, on describing possible countermeasures.

2.3.1 State Automaton

Once a malicious extension obtains the interpretive control of the UTM state automaton, it is in a good position to perform a 'Denial of Service' (DoS) attack against the UTM by simply monopolizing the thread of control and starving competing applications. Denial of service attacks assault the *Availability* property of the UTM and fall under the *Interruption* attack category in our classification scheme. This highlights the need for a mechanism for **Timing Isolation**, *i.e.* for the enforcement of some upper limit of execution 'time' allocated to untrusted extensions and for forceful preemption of delinquent code.

A different avenue for abuse of interpretive control of the UTM is to direct it in a harmful way. Realistic computer code is rarely self-sufficient and oftentimes relies on libraries and external services such as those provided by an operating system. The ability of untrusted code to transfer control to external services in arbitrary ways, however, raises the possibility

of misuse. Clever selection of external targets can be used to bypass restrictions, *e.g.* calling `panic()` to crash the machine, `spin_lock()` to monopolize or block system resources, or even transferring control to the middle of a function to circumvent parameter verification checks. Such attacks fall into the *Fabrication* category, as their effects can be equivalent to violating the system's *Authenticity* property. Preventing this type of attack requires the imposition of **Execution Control** on untrusted code and limiting its ability to direct the flow of control only to well-defined and safe entry points.

Finally, as its name suggests, the UTM state automaton is stateful itself. This compels us to consider the possibility of malicious extensions manipulating its state. Such attacks can be thought of as a formalization of the general idea of exploiting specific features of the implementation of an execution machine, *e.g.* exploiting instructions that disable interrupts, swap page table bases, *etc.* This constitutes another kind of *Fabrication* attack. It suggests the need for **Code Vetting** by the UTM state machine to monitor and restrict the use of state-changing primitives by untrusted extensions to a small set formed on the basis of a minimalistic 'need-to-use' principle.

2.3.2 Read/Write Head

Because the head serves as the main I/O abstraction of our UTM, it represents the gateway to both its scratch space (analogous to RAM) as well as its input and output (analogous to disks). However, by virtue of the fact that both application and extension share those mediums, it is possible for each of them, in particular for the untrusted extension, to gain access to the resources of the other. This constitutes an *Interception* attack on the *Confidentiality* of the system and motivates the need for the access device to implement a **Memory Isolation** policy based on the identity of the executing code.

Moreover, the write function of the head raises the specter of destructive unauthorized access to data, and the von Neumann architecture of our UTM model raises the even graver possibility of malicious code alteration. These are *Modification* type attacks against the *Integrity* of the system. They further strengthen the case for deployment of **Memory Isolation**.

2.3.3 Tape

The tape is the final building block of our UTM. Like its real-life analogues – stable storage devices – it does not play an active role in execution of the thread of control, but rather it acts as a passive medium holding raw input/output data and code. The lack of control functions obviates the need for special isolation requirements from the UTM tape.

2.4 Summary

To summarize, in this chapter we have presented a generic formal model of a computing machine and combined it with a taxonomy of security attacks. We have systematically analyzed all parts of the model with respect to the attack taxonomy and have identified all available avenues of security exposure. From the latter, we have derived a complete and exhaustive set of requirements, necessary in order to achieve comprehensive isolation among active execution contexts. For convenience, we distill that set below:

- **Memory Isolation**: prescribes the imposition of limitations on both 'read' and 'write' accesses to storage (be it real memory, or abstract tape) and forms the basis for protection among address spaces. In this way it also provides the foundation for protection against code modifications exploiting the von Neumann architecture.

- **Execution Control**: stipulates the enforcement of strict constraints over the ability of untrusted code to direct the flow of execution control. It restricts allowable branch targets only to well-defined entry points of explicitly authorized services.

- **Timing Control**: dictates the establishment of upper bounds on the permissible runtime of untrusted code and forceful preemption for violations of the constraint. Thus, timing control helps to prevent denial of service attacks and to ensure the liveness of the system.

- **Code Vetting**: prevents exploits of the stateful nature of the UTM by requiring that dangerous state transitions be identified and be made unavailable to untrusted code.

We continue with a closer look at the implementation details of various practical code isolation techniques in actual computing machines.

CHAPTER III

HYBRID CODE ISOLATION

This chapter describes the technical underpinnings of the central contribution of this dissertation, namely the proposal of a 'hybrid' approach to code isolation. We advocate the simultaneous use of a heterogeneous mix of isolation techniques, each of which is aimed at a particular isolation criterion, and whose union covers the complete set of such requirements.

Although there are many techniques that can simultaneously satisfy multiple isolation requirements, their use typically results in loss of flexibility and/or arbitrary restrictions, *e.g.*, implementation language lock-in, hardware limitations, *etc.* Hence, the key benefits of a hybrid scheme are both increased flexibility and improved performance delivered by means of alternative implementation venues.

In the remainder of this chapter, we explore the technical details of an actual hybrid prototype composed from a combination of popular individual techniques. In the first half of the chapter, we focus on the implementation and distinctive features of a few concrete homogeneous techniques. We discuss their strengths and weaknesses, and propose some improvements. In the second part of the chapter, we describe how distinct techniques can be blended together into a heterogeneous hybrid prototype with improved properties.

To better illustrate the technical points throughout the discussion, we provide simple code examples in a high-level language, as well as in low-level IA-32 assembly [47, 48]. Many of the code snippets come from a simple PPM image grayscaling code, while a larger and more realistic example is provided in Appendix B. The latter consists of the computational kernel of Edgebreaker – a state-of-the-art compression algorithm for triangle mesh 3D model descriptions. Because of its size, we focus on a subset of the Edgebreaker reference implementation in Appendix B. The complete source code is available online [87].

3.1 Homogeneous Techniques

The homogeneous techniques considered are representative of two well-known, if antagonistic, categories, namely 'software' and 'hardware' fault isolation. In the former category, restrictions are enforced through some form of software manipulation at compile-, load-, or run-time. In contrast, the latter employ features of the underlying physical hardware, which are reconfigured in such a way as to achieve the desired constraints.

3.1.1 Software Fault Isolation

The first homogeneous technique considered is generally known as 'Software Fault Isolation' (SFI). It is also known as 'sandboxing', since it can be thought of as executing unsafe extension code inside a limited sandboxed execution environment where it can do no harm. SFI was first described by Wahbe *et.al.* [102] as an alternative to hardware fault isolation and a means for reducing the communication costs experienced by software extensions at the expense of increased execution time. It operates by modifying extensions' object code to place them in a faulting domain that is logically separate from their host application. While Wahbe *et.al.*'s technique was initially described and implemented in the context of a MIPS RISC-processor based machine, CISC-processor versions have been devised in later years, such as Small and Seltzer's MiSFIT [94] and McCamant and Morrisett's PittSFIeld [60], both targeted at the Intel IA-32.

The basic idea of sandboxing is to take a potentially unsafe extension, transform its code by inserting checking and fault handling instructions interspersed wherever any kind of fault may occur, and then link and proceed to execute the extension directly as part of its host application.

In order for sandboxing to succeed, all possible sources of runtime faults must be intercepted before the fault has affected the execution environment. Many such sources exist, each violating one of the isolation requirements from Chapter 2.

Listing 1: An example usage of code sanitizing through dynamic binary rewriting. In this synthetic example, the unsafe instruction LGDT in line 3 is sanitized by rewriting it into a series of null operation NOP instructions.

```
1 movl   $0x3,%eax                    movl   $0x3,%eax
2 addl   %eax,%ebx                    addl   %eax,%ebx
3 lgdtl  $0x12345678        =>        nop; nop; ... nop        # transformation
4 movl   %eax,%ebx                    movl   %eax,%ebx
5 addl   %ebx,%ebx                    addl   %ebx,%ebx
```

3.1.1.1 Code Vetting

Ultimately, after sandboxing has been performed, extension and host code execute directly on the same hardware. Hence, there is a potential for a malicious extension to execute code that reconfigures the hardware and thus impacts its host application. Numerous such examples can be given, ranging from instructions like `LGDT`, `LLDT`, and `LIDT`, which set the bases of vitally important descriptor tables that control the CPU, to interrupt control instructions like `CLI`, `STI`, and `POPF`, which can interfere with the mutual exclusion property of critical sections. Most of these unsafe instructions have no valid use in extensions and thus should never appear in them. This makes such extensions easy to detect and deal with, by either rejecting them outright or 'sanitizing' the unsafe instructions.

The details of these approaches necessarily depend on the form that extensions take. If distributed as source code, extensions can be compiled by a trusted code generator on the fly and thus be code vetted by construction. If distributed in binary form, code vetting can be achieved by disassembling extensions' object code and verifying the absence of unsafe instructions. A more powerful and complex process of code vetting by 'sanitizing' unsafe instructions is through dynamic binary rewriting, a technique capable of modifying and/or inserting object code into an already compiled binary, essentially turning unsafe code into null operations. Listing 1 provides a visual example of how dynamic binary rewriting could be used to sanitize potentially unsafe instructions into safe, empty ones.

3.1.1.2 Memory Isolation

Because host and extension share the execution environment, the need arises for the former to protect its data from the latter. Hence, in-core data stored in random access memory

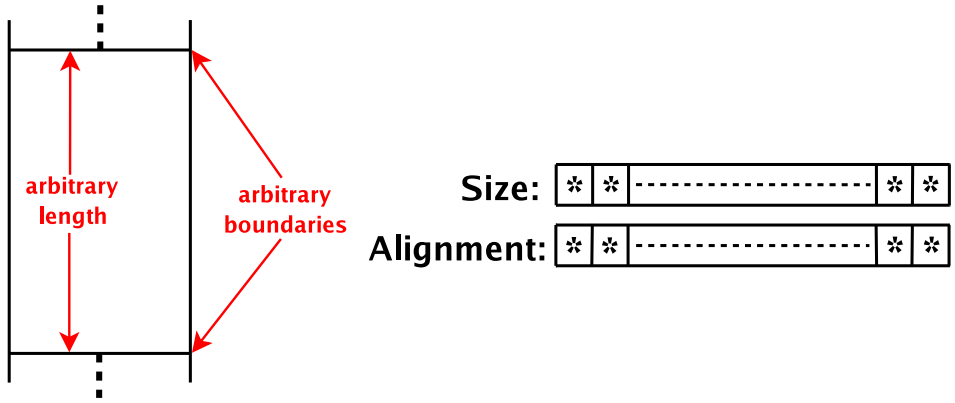

Figure 5: Memory segment restrictions for the sandboxing code from Listing 2.

Listing 2: Memory access verification code that can deal with the most general form of IA-32 indirect addressing as well as with arbitrary (byte granularity) memory segment boundaries and sizes. Lines 1–7 sandbox the memory reference on line 8.

```
1 pushl   %eax                  # save temporary register
2 leal    0x1(%ebx),%eax        # compute effective address
3 cmpl    %eax,$LO_BOUND        # compare to low segment bound
4 jb      out_of_bounds         # handle low bound violations
5 cmpl    %eax,$HI_BOUND        # compare to high segment bound
6 jae     out_of_bounds         # handle high bound violations
7 popl    %eax                  # restore temporary register
8 movzbl  0x1(%ebx),%eax        # original instruction
```

is vitally important to any isolation scheme. Lacking any memory protection measures, in-core data is directly accessible to whichever code controls the CPU at the moment.

The general approach for sandboxing memory isolation is to separate each extension into its own contiguous logical memory segments and then to inspect and possibly modify its object code in such a way as to prevent unauthorized external memory references. Each segment is assigned separate access rights. Code segments, for example, can be either readable or not, but are always immutable in order to guard against self-modifying code attacks. In contrast, data segments can be readable, writable, or both, but cannot be the target of execution control transfers, to prevent execution of synthetic code.

In order to verify each memory reference for conformance to these restrictions, however, its target must be computed. In the case of direct addressing, this is trivial because the

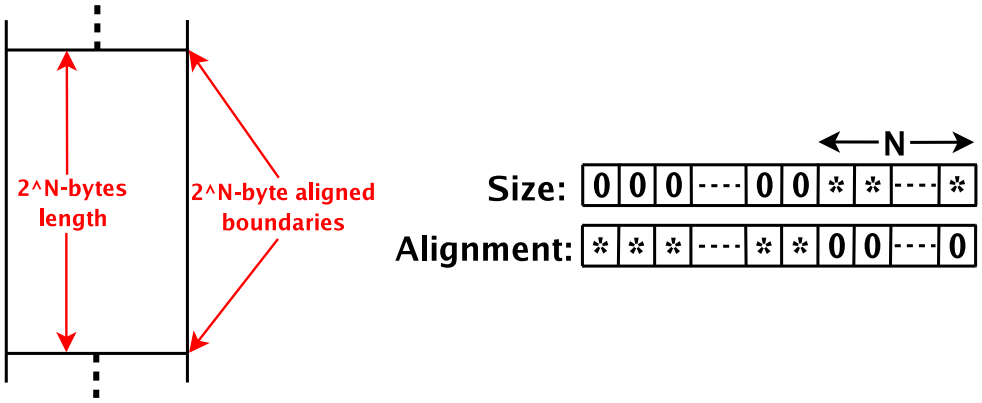

Figure 6: Memory segment restrictions for the isolation code from Listing 3.

Listing 3: Faster memory access verification code that can also deal with the most general form of IA-32 indirect addressing, but requires memory segments of 2^N byte size and alignment. Lines 1–6 sandbox the memory reference on line 7.

```
1 pushl   %eax                  # save temporary register
2 leal    0x1(%ebx),%eax        # compute effective address
3 andl    $BIT_MASK,%eax        # mask-off low-order bits
4 cmpl    $SEGMENT_ID,%eax      # compare high-order bits
5 jne     out_of_bounds         # handle violation
6 popl    %eax                  # restore temporary register
7 movzbl  0x1(%ebx),%eax        # original instruction
```

fixed target is readily available as part of the binary instruction code. Hence, direct addressed memory references can be verified at a low one-time cost during the loading stage of the extension and will not incur any further runtime overhead. The targets of indirect-addressed memory references, however, cannot be known at load time because they are variable and depend on the values of the registers used in the address calculation at the moment of execution. Therefore, sandboxing indirect addressed memory references requires the insertion of dynamic checks into the binary.

The most general form of indirect addressing allows the computation of the effective-address to include a base address, an offset, and a scaled index combined according to the formula:

$$BASE + R_{offset} + R_{index} * SCALE$$

Expressed as:

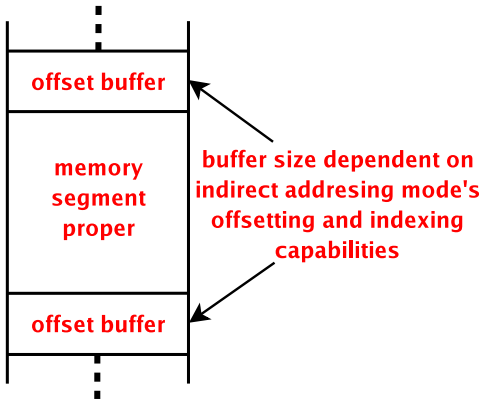

Figure 7: Memory segment layout with top and bottom guard buffers. Used with the sandboxing code from Listing 4.

$$BASE(R_{offset}, R_{index}, SCALE)$$

in the GNU Assembler syntax, also known as AT&T syntax, that is used throughout this document.

The IA-32 instruction set requires the presence of at least one argument of the sum, but allows any permutation in which all or some of the remaining arguments are omitted. It offers a lot of flexibility by permitting any CPU register, including word and byte sub-registers like `%ax`, `%ah`, `%al`, *etc.*, to be used in the offset and index, but it limits the scale factor of the index to the numbers 1, 2, 4, or 8. The base can be any number, but both the base and the index scale must be fixed and known at compile time because they are encoded into the binary representation of the instruction.

The implementation of memory reference sandboxing involves a compromise between two interrelated parameters: the alignment and size restrictions on the choice of memory regions on the one hand, and the speed and efficiency of the associated sandboxing code on the other hand. Implementations trade flexible choice of memory segments' parameters for optimized verification code as shown in Figure 6 and Listing 3, or vice versa as in Figure 5 and Listing 2.

In the interest of completeness, we should mention another alternative that is aimed at avoiding the need to compute the effective-address of the target. It relies on allocating exclusion buffer zones adjacent to each memory segment that are big enough to cover

Listing 4: (a) A fixed base is statically verifiable at load time, (b) operates on buffered, 2^N byte sized/aligned segments and eliminates effective-address computation but still requires saving/restoring of the base address due to a destructive check, (c) operates on buffered segments of arbitrary size/alignment and eliminates the need for both effective-address computation and for saving/restoring the base address, thanks to a non-destructive check.

```
1  # (a)
2  movl    $0x3,(%eax,%ebx,4)          # original instruction
3
4  # (b)
5  pushl   %eax                        # save temporary register
6  andl    $BIT_MASK,%eax              # mask-off low-order bits
7  cmpl    $SEGMENT_ID,%eax            # compare high-order bits
8  jne     out_of_bounds               # handle violation
9  popl    %eax                        # restore temporary register
10 movl    $0x3,(%eax,%ebx,4)          # original instruction
11
12 # (c)
13 cmpl    %eax,$LO_BOUND              # compare to low segment bound
14 jb      out_of_bounds               # handle low bound violations
15 cmpl    %eax,$HI_BOUND              # compare to high segment bound
16 jae     out_of_bounds               # handle high bound violations
17 movl    $0x3,(%eax,%ebx,4)          # original instruction
```

the largest offset computable with that instruction set's indirect addressing modes. Then, the validity of each indirect memory access can be determined by verifying *only* that its base address falls within the original memory segment, as any possible offset will at most displace it within the safe buffer zones. Examples of this technique are shown in Figure 7 and Listing 4.

Given the IA-32 architecture's considerable freedom in the formation of indirect mode effective-addresses and the large size of the registers that can be used as offsets and indices, the buffered approach would waste a significant amount of memory as guard buffers. Allowing for 32-bit offset and index registers would actually require guard buffers covering the whole address space. Therefore, this approach is not well suited to the IA-32 architecture and is only appropriate on platforms with more restricted indirect addressing modes.

Furthermore, there is a simple optimization that can be applied equally well to all sandboxing techniques. Similar to a compiler optimization, it reduces the cost of sandboxing by custom tailoring the verification code in the simpler cases of indirect mode addressing

Listing 5: Generic optimizations for sandboxing memory isolation code applicable for simple use of indirect addressing mode where the effective-address is pre-computed in a single register. (a) arbitrary memory regions, (b) 2^N byte sized/aligned memory regions.

```
1  # (a)
2                                  # ELIMINATED: save temporary register
3                                  # ELIMINATED: effective address computations
4  cmpl    %edi,$LO_BOUND                # compare to low segment bound
5  jb      out_of_bounds                 # handle low bound violations
6  cmpl    %edi,$HI_BOUND                # compare to high segment bound
7  jae     out_of_bounds                 # handle high bound violations
8                                  # ELIMINATED: restore temporary register
9  movb    %dl,(%edi)                    # original instruction
10
11
12 # (b)
13 pushl   %edi                          # save temporary register
14                                 # ELIMINATED: effective address computations
15 andl    $BIT_MASK,%edi                # mask-off low-order bits
16 cmpl    $SEGMENT_ID,%edi              # compare high-order bits
17 jne     out_of_bounds                 # handle violation
18 popl    %edi                          # restore temporary register
19 movb    %dl,(%edi)                    # original instruction
```

where the effective-address requires no computation and is already easily accessible in a platform register. An example of this optimization is detailed in Listing 5. Depending on the memory segment model, it can eliminate from 1 to 3 instructions from each verification instance resulting in approximately 40% cost reduction (3 out of 7 instructions) in those cases.

3.1.1.3 Execution Control

Every time a host application makes use of an extension, it necessarily relinquishes, albeit temporarily, control of the execution environment to that extension. The latter is then generally able to direct the flow of control in an arbitrary way. This fact necessitates the imposition of flow control restrictions on software fault-isolated extensions in order to contain them within the confines of their sandboxes.

As illustrated in Figure 8, flow control falls into two categories: *external*, which takes the form of callbacks into the host application or OS and is typically used to obtain services needed by the extension, and *internal*, which implements coding logic and takes the form of intra-extension jumps, branches, and calls.

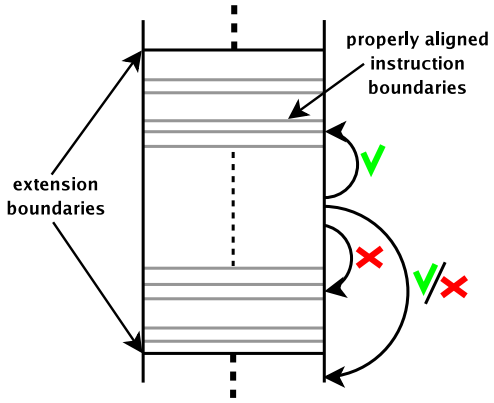

Figure 8: Execution control restricts external callbacks only to explicitly allowed service entry points and also guards against internal misaligned instruction decoding attacks such as the one shown in Listing 7.

Listing 6: An example of a misaligned callback attack. It shows how, if allowed to invoke an arbitrary callback entry point, an extension can circumvent protection mechanisms, or even invoke dangerous external symbols, *e.g.*, `panic()`, *etc.*

```
1                          int service_callback(int extID, ...)
2                          {
3 normal entry point  ->     if(!privileged(extID))
4                                    return ERROR_NO_PRIVILEGE;
5
6 spoofed entry point ->     /*
7                             * Service code...
8                             */
9
10                         return 0;
11                         }
```

Restrictions on the flow of control are needed for both categories. In the former, the unchecked choice of callback entry points could allow a malicious extension to circumvent parameter checks (see Listing 6) and could lead to disastrous results ranging from security violations to crashes. In the latter, a malicious extension running on a variable instruction length architecture could exploit misaligned instruction decoding and circumvent code vetting and execute arbitrary code with disastrous results.

Code vetting guarantees that the machine instructions that make up a sandboxed extension do not include any that can impact the execution environment in an unsafe way. On fixed instruction set architectures, the flow of control can only be directed at the fixed

Listing 7: An example of a misaligned instruction decoding attack. It shows how an otherwise innocuous instruction (line 3) can be decoded as malicious code halting the machine (lines 7–8) through a misaligned branch to its tail fraction.

```
1  # Offset   Object-Code          Insn    Operands
2
3      0:    b8 90 90 f4 fa       mov     $0xfaf49090,%eax
4      5:
5
6
7      0:    b8 90 90             .byte   0xb8, 0x90, 0x90
8      3:    f4                   cli                          # DANGER!
9      4:    fa                   hlt                          # DANGER!
10     5:
```

instruction boundaries, so there is no room for instruction misinterpretation.

On variable instruction length architectures, however, the flow of control can be directed at a finer byte granularity. This makes it possible to construct machine code in such a way that it decodes as innocuous instructions when accessed properly aligned, thus allowing it to slip under the radar of code vetting, but decode as unsafe instructions when accessed misaligned. An example of such code for the IA-32 architecture is shown in Listing 7.

The concrete implementation details of internal flow control are both similar and different among techniques. The similarity is that all techniques rely on rewriting the binary and the injection of additional target verification code before each indirect branch. The more important differences, however, lie in the way in which the set of allowable branch targets is determined.

Execution Control in MiSFIT

MiSFIT [93, 94] is one of the first software fault isolation tools. Developed by Small and Seltzer at Harvard University, it was designed to operate as a filter stage to the assembly output of `gcc`, the C compiler that is part of the GNU Compiler Collection [27]. Because it operates on source code, albeit low-level assembly source code, MiSFIT is able to glean and exploit semantic information about the set of valid control flow targets in a program, which is lost after the final compilation. It builds a table of legitimate targets by enumerating all the labels and procedure entry points in the intermediate assembly and then injects sandboxing code before each indirect control transfer instruction that checks the actual

runtime target for presence in the table.

To limit the table search time in the face of a potentially large number of valid target addresses, MiSFIT implements the table as a sparse, open addressed hash table, claiming near-linear search time [94]. The table is stored as an array where the hash of the key (target address) yields an index into the array to check. Effectively, the hash bucket size is 1, and collisions are handled by overflowing into the next bucket. That is, if an item hashes to slot n that is already occupied, the insertion algorithm attempts to successively place it in $n + 1$, $n + 2$, *etc.*, until it succeeds. An advantage to this open addressed hashing is that if the original slot fails to produce a match during a lookup, the next few successive slots to be checked are likely to already be present in the cache because they are adjacent. Thus, even if a probe fails, the cost of subsequent probes is reduced. Furthermore, the arrangement allows one to trade table size *vs.* lookup speed because increasing the size of the hash table decreases its density. Assuming a hashing function that spreads the items around evenly, in theory the number of probes per lookup can be reduced to near-constant. As noted by Small *et.al.*, an evenly distributed table with a density of 50% will require an average of 1.5 probes per lookup. This is important as indirect control transfers are quite common, constituting the implementation of choice for `switch` statements, function pointers, object dispatch tables, and procedure returns. Listing 8 provides an illustration of the code transformations MiSFIT performs in order to instrument the indirect control transfers in a sandboxed binary.

MiSFIT is programming language agnostic, as it is essentially an additional back-end stage of the compiler. For the same reason, however, it suffers a drawback in that it is not applicable to legacy or proprietary extensions for which source code may not be readily available. Furthermore, MiSFIT's approach presents a trade-off between the size of the hash-table and the speed of control transfer target verification.

Execution Control in PittSFIeld

PittSFIeld [59, 60] is a software fault isolation tool for the IA-32 architecture developed by McCamant and Morrisett at MIT. Like MiSFIT, it takes the form of a filter for `gcc` generated assembly, but unlike it, it employs a sharply contrasting approach to sandboxing.

Listing 8: MiSFIT code transformations for sandboxing of indirect control transfers. (a) original indirect control transfer, and (b) sandboxed by MiSFIT (code extracted from the MiSFIT 0.2 distribution).

```
1  # (a)
2          call   * %edi
3            .
4            .
5            .
6
7  # (b)
8          movl     %edi,%eax
9          call     misfit_ind_call_from_graft
10           .
11           .
12           .
13 misfit_ind_call_from_graft:
14         pushl    %esi
15         movl     %eax,%esi                # save original target addr
16
17         cmpl     $0,%eax                  # jumping to NULL?
18         je       3f
19 1:
20         movl     misfit_hashtable,%ecx    # get base of table
21         andl     misfit_hashmask,%eax     # and in mask
22         addl     %eax,%ecx                # load the test word
23         cmpl     (%ecx),%esi              # is this the one?
24         jne      2f                       # nope, try again
25         movl     %esi,%eax                # restore saved esi
26         popl     %esi
27         jmp   *  %eax                     # call it
28 2:
29         cmpl     $0,(%ecx)                # have we hit a zero?
30         je       3f                       # yep, fail
31         addl     $4,%eax                  # nope, try next slot
32         jmp      1b
33 3:
34         pushl    %esi
35         call     misfit_bad_ind_call
```

Whereas the philosophy of MiSFIT is to accept and cope with the variable instruction length nature of the architecture, PittSFIeld's approach is to partially restrict it to a fixed instruction length architecture. Its basic strategy is to enforce artificial alignment restrictions by regarding memory as a series of 'chunks', which satisfy the following restrictions:

1. The size and alignment of each chunk are fixed at a power of 2 bytes, equal to or larger that the length of the longest instruction encoding, *e.g.*, $2^4 = 16$ bytes or larger for IA-32.

2. No instruction can straddle a chunk boundary.

3. Instructions, which could be the target of a control transfer, must always be placed at the beginning of a chunk.

To satisfy those criteria, PittSFIeld intersperses no-op filler instructions as needed to maintain proper alignment. This is always possible as the null `NOP` instruction is conveniently 1 byte sized. Nevertheless, an optimizing compiler like `gcc` can also utilize instructions with longer encoding, which have no semantic impact on the code, such as `leal 0(%esi),%esi` for example.

The padding transformation illustrated in Listing 9 allows PittSFIeld to treat the instruction stream both as a sequence of variable length instructions *and*, for the purposes of control transfers, as a sequence of fixed 2^4 byte macro-instructions. The artificial alignment of control transfer targets simplifies the verification code that needs to be injected before each indirect control transfer. Target alignment check or enforcement is reduced to comparing or coercing of the 4 least-significant address bits to 0, whereas the 2^n byte memory sandbox limits are checked or enforced by comparing or coercing the $(32 - n)$ most-significant address bits to its common address bit-prefix 'tag'.

The price for this simplification is paid in terms of the code inflation caused by the padding transformation. Although semantically neutral, the padding code induces "bubbles" in the execution pipeline of IA-32 super-scalar processors and also reduces the effectiveness of the instruction cache by wasting some of its capacity on filler code.

Listing 9: PittSFIeld padding code transformations for sandboxing of indirect control transfers. (a) short test function (54 bytes), and (b) sandboxed by PittSFIeld (110 bytes, processed with the PittSFIeld 0.2 distribution).

```
1  # (a)
2
3  .globl main
4          .type   main, @function
5  main:
6          leal    4(%esp), %ecx
7          andl    $-16, %esp
8          pushl   -4(%ecx)
9          pushl   %ebp
10         movl    %esp, %ebp
11         pushl   %ecx
12         subl    $4, %esp
13         movl    (%ecx), %eax
14         movl    apf(,%eax,4), %eax
15         movl    %eax, fp
16         movl    fp, %eax
17         movl    $3, (%esp)
18         call    *%eax
19         addl    $4, %esp
20         popl    %ecx
21         popl    %ebp
22         leal    -4(%ecx), %esp
23         ret
24         .size   main, .-main
25
26
27
28
29
30
31
32
33
34
35
36
37
```

```
# (b)

.globl main
        .type   main, @function
        .p2align 4
main:
        leal    4(%esp), %ecx
        andl    $-16, %esp
        pushl   -4(%ecx)
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ecx
        .p2align 4
        subl    $4, %esp
        movl    (%ecx), %eax
        movl    apf(,%eax,4), %eax
        .p2align 4
        movl    %eax, fp
        movl    fp, %eax
        .p2align 4
        movl    $3, (%esp)
        movl    %eax, %ebx
        .p2align 4
        mov     %esi, %esi
        andl    $0x20ffffff, %esp
        andl    $0x10ffffff0, %ebx
        call    *%ebx
        addl    $4, %esp
        popl    %ecx
        popl    %ebp
        andl    $0x20ffffff, %ebp
        leal    -4(%ecx), %esp
        .p2align 4
        andl    $0x20ffffff, %esp
        andl    $0x10ffffff0, (%esp)
        ret
        .size   main, .-main
```

Like MiSFIT, PittSFIeld is also programming language agnostic and operates as a filter in the back-end of the compiler. Thus, it also shares the related drawback with respect to legacy and proprietary code. Unlike MiSFIT, however, it does not present a memory *vs.* speed trade-off. Control transfer target checks are always fast, but at the expense of increasing the extension's code, as opposed to increasing its data.

An Improved Proposal

A shared drawback to the previous two approaches is that, because of the 'filter' nature of their implementations, they can only be applied to extensions distributed in source and compiled with a specific compiler. This renders them unusable for the important classes of legacy and proprietary extensions. We now present a novel variation to the basic MiSFIT design that aims to overcome this restriction and to adapt the technique for use with already pre-compiled machine code while maintaining rapid lookup speed.

The crux of the problem lies in the fact that after compilation, the semantic information about the set of valid control transfer targets, *i.e.*, the set of all labels and function entry points, is lost. We observe, however, that for purposes of software fault isolation, we do not need full semantic information. In fact, in order to ensure that misaligned control transfers do not occur, all that is required is a map that describes the boundaries of all well-aligned instructions and that supports rapid lookups. We propose a packed bitmap implementation with a single bit per byte of extension code. The complete bitmap then represents a boolean function providing exactly the needed description.

The size complexity of such a bitmap is $O(n)$, or linear, in the size of the extension code. In fact, because each bitmap byte represents 8 bytes of code, the total bitmap size is fixed at 1/8, or 12.5%, of the size of the extension code. The computational complexity of creating the bitmap is also linear in the size of the code, because it involves disassembling the well-aligned instruction stream, itself easily reduced to instruction by instruction constant time lookups in a table describing the architectural machine code. Finally, a lookup in the bitmap table itself is also $O(1)$, or constant time, and benefits from the IA-32's 'bit test' instruction. An example of the bitmap lookup code is provided in Listing 10.

Thus, the proposed new design allows us to achieve fast constant time lookup, as well

Listing 10: Example code for indirect control transfer target verification employing a novel bitmap lookup table. It encodes a boolean function that describes all well-aligned instruction boundaries and enables constant time lookups. (a) illustrates a typical indirect control transfer instruction implementing a dispatch table, (b) illustrates the same instruction along with bitmap-based isolation code. Note how in line 6 the conversion from absolute target address to an offset within extension code is folded into effective-address computation once at isolation time. CODE_BASE and BITMAP_BASE represent the respective numerical addresses.

```
1  # (a)
2          call   * APP(,%edx,4)
3
4  # (b)
5          pushl    %eax
6          leal     APP-CODE_BASE(,%edx,4),%eax
7          cmpl     $MAX_OFFT,%eax
8          ja       out_of_bounds
9          bt       %eax,BITMAP_BASE
10         jz       misaligned
11         popl     %eax
12         call   * APP(,%edx,4)
```

as a dense but fixed table size that is likely shorter than a sparse MiSFIT hash-table of equivalent lookup speed.

3.1.1.4 Timing Control

The final requirement for complete software fault isolation is timing control. It refers to the need to impose an upper limit on the execution time of extensions and is dictated by the loss of control implied in each extension invocation and the potential for mounting 'denial of service' attacks.

Formal techniques have been proven unable to deal with the issue of timing control, as in general, it is equivalent to solving the NP-complete Halting Problem. Existing software fault isolation systems also do not provide a software solution. Instead, they rely on the traditional OS approach of detection and preemption through the means of hardware timers. The reasons for this reduce to the inefficiency of software implementations.

For example, binary rewriting techniques could be employed to enforce an upper limit on runtime by interspersing timing checks throughout the binary. While implementable in practice, this is costly, because in order to guarantee preemption, a check must be inserted

in each basic block of the extension's code. However, basic blocks are fairly short, especially on CISC architectures such as the IA-32, where a basic block's typical length is about 2 or 3 instructions and branches occur about every 6 instructions [36]. This makes the overhead of an SFI timing control solution too high to be practical and motivates the use of timers and preemption.

3.1.2 Hardware Fault Isolation

Hardware fault isolation is an alternative class of homogeneous code isolation techniques that take advantage of features built into modern processors and chipsets. Such features typically include hardware timers, processor support for memory segmentation, and sometimes privilege levels, though different platforms support different sets of features. In particular, the Intel IA-32 platform supports all of the above, though segmentation is notably missing from EM64T, its 64-bit extension. While the latter may obviate some of the specific hybrid examples presented later in this chapter, it does not negate the larger premise of this dissertation about the usefulness of hybrid approaches in general.

Hardware-based techniques are not unique to code isolation. In fact, such techniques are widely used in software ranging from the kernels of ordinary consumer operating systems like Linux [57] or Windows [90], to research isolation and extensible kernels such as Denali [107], Exokernel [22], and Palladium [16], to virtualization solutions like Xen [10], VMware [101], and Microsoft Virtual PC [18, 62]. Irrespective of their purpose, however, all of the systems above employ similar hardware features to enforce the boundaries between software components. We continue by taking a closer look at the technical implementation details of what that entails.

3.1.2.1 Paging and Segmentation

Paging and segmentation are two popular and commonly available memory management features of modern processors. While the latter is a mechanism specifically designed for isolating individual software modules so that many of them could run on the same processor without interfering with each other, the former is meant as a mechanism for implementing traditional demand-paged virtual memory. Nevertheless, given the right configuration, a

paging system can also be used to isolate software modules. In fact, that is what modern commodity operating systems employ, because paging is practically universally available in processors today and thus affords the greatest portability.

In terms of our classification system, either paging or segmentation can serve as a vehicle for enforcing the dual *memory isolation* and *execution control* requirements. The reason for this is that both paging and segmentation are implemented as address transformations that the CPU applies universally to all addresses touched by it during execution in protected mode. Thus, either technique can be used equally well to control read/write data accesses, *i.e.* memory isolation, as well as instruction fetch accesses, and thus the boundaries of control flow transfers, *i.e.* execution control.

IA-32 programs operate in 'logical' address spaces, set up for them by the operating system and described by a set of segment selectors or a page table, respectively. At every memory access, the CPU converts the logical address to be accessed into a physical one to be emitted on the external address bus. The conversion takes place in the context of the currently active address space, with the segmentation system mapping the logical address to a linear address, and the paging system mapping the latter into the final physical address as shown in Figure 9.

Segmentation

Under IA-32 segmentation, logical addresses are tuples consisting of a segment selector, ofter an implicit default, and an offset. As shown in detail in Figure 10, a segment selector's value specifies a table of segment descriptors and an index into it. Each table entry defines a region of memory by means of its base address and length, as well as some associated properties, such as whether its contents can be executed, read, or written. The number of segments that can be defined is only limited by the size of the descriptor tables and is very large. The segments that can be used simultaneously, however, are limited to the much smaller number of segment selector registers that are used to pick out a descriptor for each memory access. The IA-32 architecture has 6 of those, and it typically uses one (`CS`) as the context of the code it executes, another (`SS`) for the current stack, and the remaining four (`DS`, `ES`, `FS`, and `GS`) to refer to data.

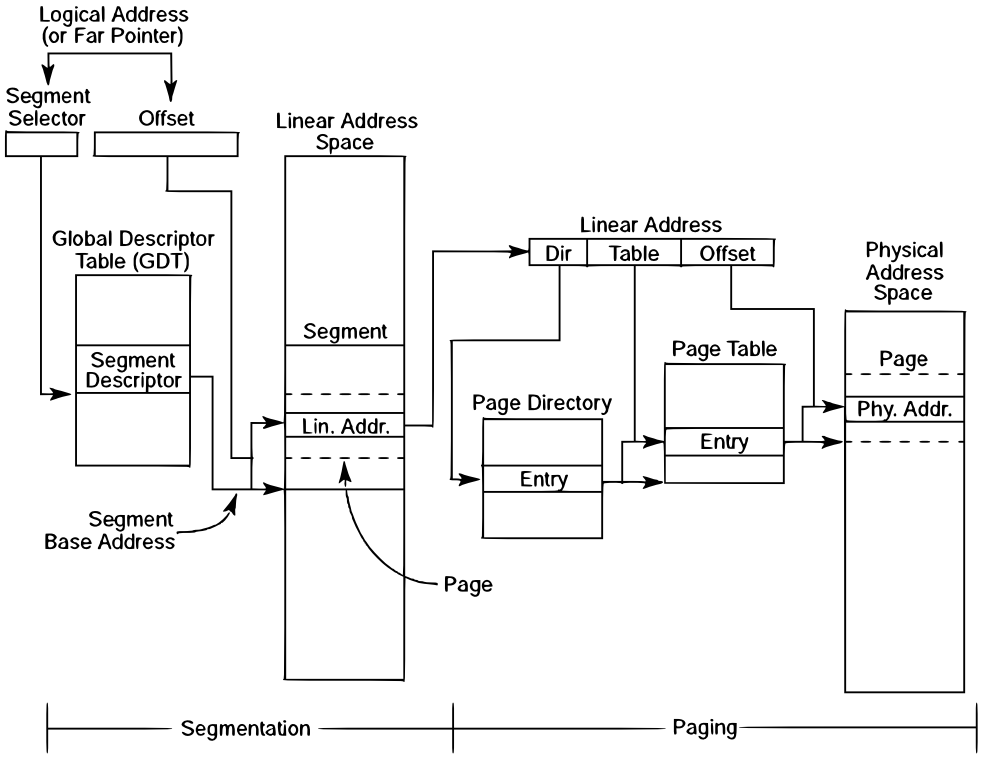

Figure 9: Segmentation and paging mechanisms. Source: *IA-32 Intel ® Architecture Software Developer's Manual, Volume 3: Architecture and Programming Manual.*

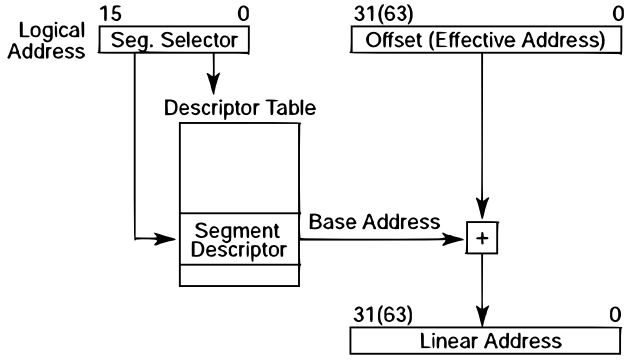

Figure 10: Logical to linear address translation. Source: *IA-32 Intel ® Architecture Software Developer's Manual, Volume 3: Architecture and Programming Manual.*

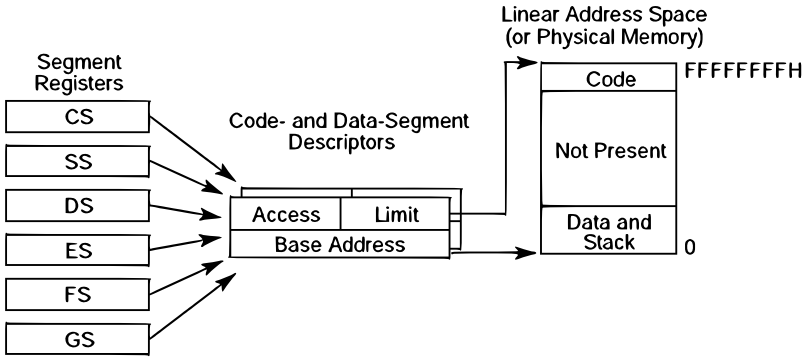

Figure 11: Unprotected flat memory model. Source: *IA-32 Intel ® Architecture Software Developer's Manual, Volume 3: Architecture and Programming Manual.*

Those default roles, however, can be altered by means of *segment override* instruction prefixes, through which, for example, one can attempt control transfers to a data segment, or reads/writes to a code segment. To prevent misuse of memory and to enforce protection among address spaces, the CPU always checks each access against the protection bits and the boundaries of its target segment descriptor. Disallowed and out-of-bounds accesses are caught and result in an exception and the transfer of control back to the OS, which in turn is responsible for taking appropriate measures against the misbehaving software module.

The described segmentation facility is flexible, and allows for the implementation of many alternative system designs. On one end of the spectrum, overlaying read-only code with read/write data segments, as shown in Figure 11, can result in essentially unprotected designs that make minimal use of segmentation. On the other end of the spectrum, mutually exclusive code and data layouts can afford protection along with a varying level of address space complexity, ranging from simpler flat models as in Figure 12, to elaborate multi-segment models as in Figure 13.

Despite being broadly available, hardware support for segmentation is not universal. This has caused it to be utilized mostly in research prototypes of extensible systems. Numerous such examples have been developed, notably the the L4 microkernel [38, 55], the Palladium system [16], and kernel plugins [29], with the latter two built on top of the classic monolithic Linux kernel.

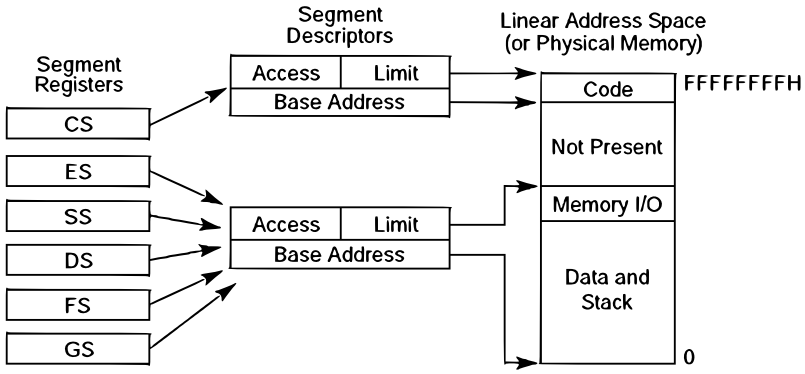

Figure 12: Protected flat memory model. Source: *IA-32 Intel ® Architecture Software Developer's Manual, Volume 3: Architecture and Programming Manual.*

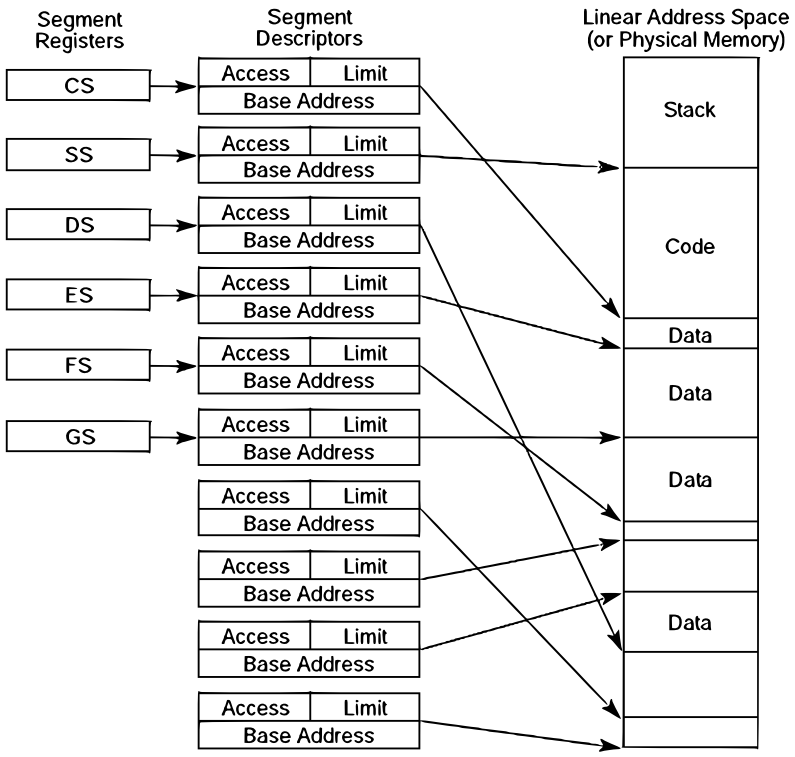

Figure 13: Multi-segment memory model. Source: *IA-32 Intel ® Architecture Software Developer's Manual, Volume 3: Architecture and Programming Manual.*

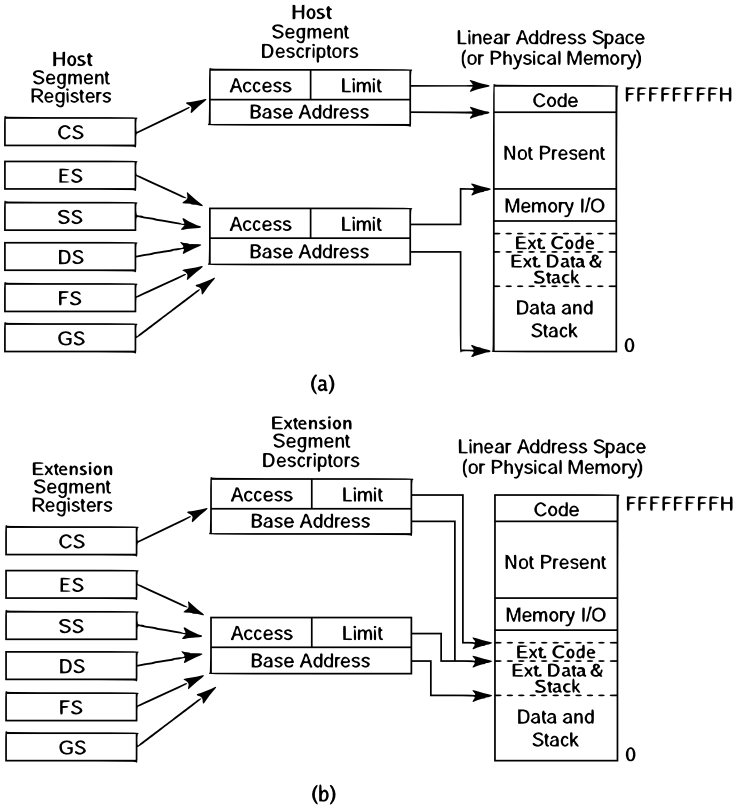

Figure 14: Host/extension segmentation configuration. (a) describes the runtime environment when the host software module is operating, and (b) describes it when the extension is operating. Note how the extension's memory segments are a proper read/write-able subset of its host's, signifying the one-way control relationship between them. Note also the level of indirection between the control registers and the actual segment descriptions allowing for rapid address space switching.

Typically, the entity being extended is described by a set of segments in a protected memory model, *i.e.*, with one or more non-overlapping code and data segments and the appropriate permissions, usually read/execute for the code and read/write for the data. The extension itself is described by a similar, but separate set of segment descriptors with their actual memory backing store as a proper subset of the controlling host modules' data as shown in Figure 14.

There is an inherent level of indirection between the segment selector registers that control the 'context', in which memory accesses are evaluated, and the segment descriptors that actually establish their bounds and permission bits. This indirection is a key feature of segmentation-based schemes, because it allows them to perform rapid address space switches

by only reloading the selector registers' values explicitly, and the relatively small descriptors implicitly by the hardware. Compared to the costs of address space switches by means of paging, segmentation offers a much faster alternative.

Paging

The IA-32 architecture's paging system maps the linear address space, output from the segmentation system, into external physical memory. The address space is split into fixed size 'pages' of typically 4 KB each, though larger 2 MB or 4 MB pages are also supported. Each linear address space page is mapped individually. It can be either left unmapped, or mapped onto a 'frame' of physical memory, or mapped onto external disk storage. This permits system software to emulate the presence of 'virtual' memory much larger than the available physical RAM by 'demand-paging' a smaller working set. If the page containing a linear address is not present in memory, the CPU generates a page fault. The OS exception handler then allocates a frame, loads its contents from disk, and maps the linear address to it.

Paging differs from segmentation in a number of aspects. Segments can vary in size, but are inherently comprised of a range of continuous addresses with no holes and typically encompass completely the object they contain, *i.e.*, code or data. Pages, on the other hand, are always of a fixed predetermined size and can have arbitrarily scattered mappings that can result not only in logically continuous but physically discontinuous objects, but also in structures that are partially in memory and partially on disk.

Figure 15 shows the linear to physical address conversion process and the multi-level hierarchical mapping structure that controls it. Page mapping descriptors are packed into memory pages themselves. The page at the top level of the hierarchy, or the page directory, contain Page Directory Entries (PDEs), which are special mappings that point to pages in the lower level of the hierarchy, or the page table. Page table pages contain Page Table Entries (PTEs), which in turn point to the actual ordinary data pages.

Aside from a target address, both PDE and PTE map entries also contain a 'present' flag, as well as 'read/write' and 'user/supervisor' flags. While these flags are useful for the implementation of demand paging and protecting the OS kernel from a simultaneously

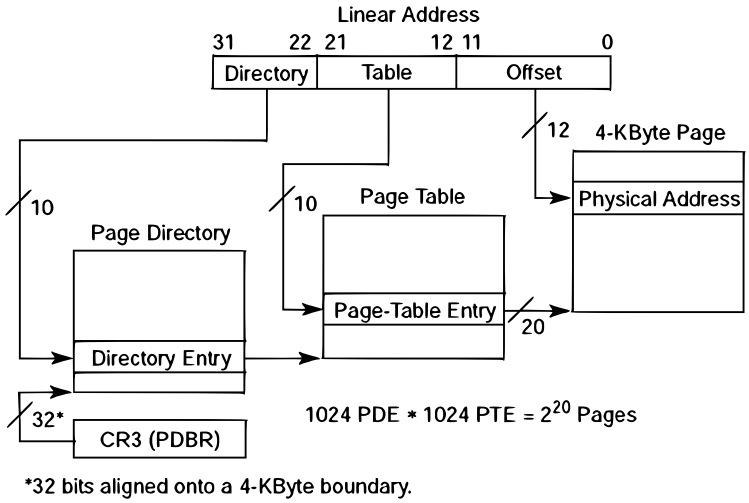

Figure 15: Linear to physical address translation. Source: *IA-32 Intel ® Architecture Software Developer's Manual, Volume 3: Architecture and Programming Manual.*

mapped process, the means for implementing multiple address spaces are provided by the ability of system software to replace the mapping structures entirely. The setup commonly used today is for the OS to build a separate page table for each process, each describing the (static) pages of the OS kernel and of its process. The reason for this arrangement, is the high cost of page table switching. With a 4 byte PTE that typically maps 4 KB of memory, we obtain a page table overhead ratio of 1/1024. Thus, 1 GB of RAM, a reasonable number by today's standards, would require 1 MB of page tables.

To speed up the mapping process, modern CPUs cache mappings in a special built-in cache memory known as a Translation Lookaside Buffer or TLB. Given increasing memory sizes (and hence TLB sizes) and the widening gap between CPU and main memory speeds, the cost of flushing and repopulating the TLB during a page table switch can easily reach many thousands of wasted clock cycles [99], and thus be prohibitively expensive unless amortized over a relatively long period of execution in the new address space.

3.1.2.2 Privilege Levels

Privilege levels, also known as 'modes of execution', are a hardware approach for enforcing specific restrictions on the types of instructions that code running on the CPU can execute.

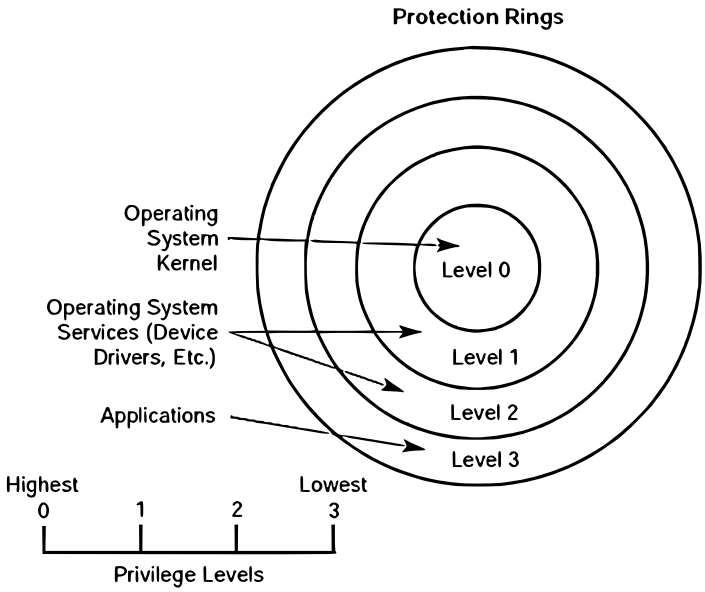

Figure 16: Privilege Rings. Source: *IA-32 Intel ® Architecture Software Developer's Manual, Volume 1: Basic Architecture.*

Typically, CPU hardware supports two privilege levels or modes, supervisor mode and user mode, though some chips support more, *e.g.*, the IA-32 provides four levels of increasing privileges as shown in Figure 16.

At any given point in time, the CPU is running at one particular privilege level, and transitions between levels that elevate privileges can occur only through entry points defined in advance by the most privileged system software. In addition, there is a different set of instructions available at each privilege level, with higher privilege corresponding to larger sets including more powerful system control instructions. Software executing at the highest privilege has full control over the hardware and responsibility for the setup of the privilege system.

The IA-32 segmentation system requires that each segment be assigned a specific descriptor privilege level, or DPL in Figure 17. Code executing within that segment cannot exceed its privilege, though it could run at a lower one in the case of a conforming code segments invoked from a less privileged one. As the DPL is specified as a two bit number, it allows for the assignment of any of the IA-32's 4 privilege levels to any segment. This,

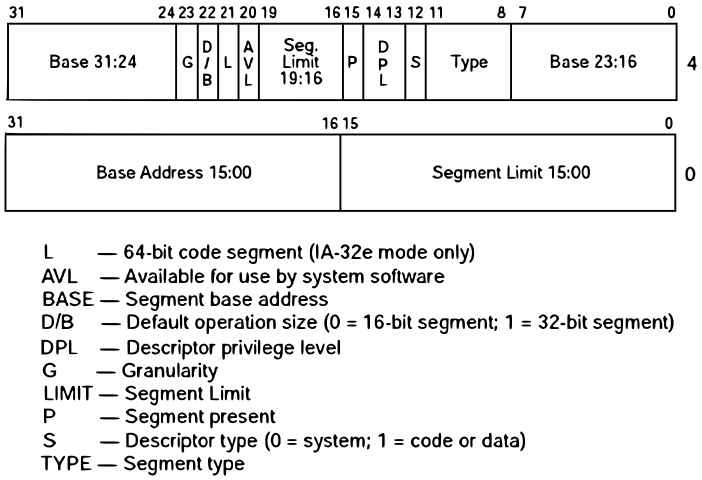

Figure 17: Segment Descriptor. Source: *IA-32 Intel ® Architecture Software Developer's Manual, Volume 3: Architecture and Programming Manual.*

however, is not the case with paging.

The IA-32 paging system also requires that each page be assigned a specific privilege level in its page table entry. However, the PTE only allocates a single bit for the privilege specification, see Figure 18, so paging is restricted in its ability to utilize privilege levels. In effect, paging compresses the IA-32's 4 privilege levels into two, with levels 0, 1, and 2 mapping to 'supervisor', and level 3 mapping to 'user'.

Irrespective of the number of available levels, privilege level hardware can be exploited to implement the code vetting requirement for isolation by simply placing untrusted extension code at a level low-enough so that it will not have access to potentially dangerous instructions, such as, *e.g.*, `CLI`, `LGDT`, or `LDS`, which respectively disable interrupts, load a new global segment descriptor table, or reload a segment selector register on IA-32.

The chief benefit of hardware enforced code vetting lies in the fact that it decouples runtime overhead from the type and quantity of the instruction mix. Because code vetting is in essence 'folded' into the protection checks already performed by the privilege level hardware, it is essentially free of per-instruction runtime overhead.

In turn, the drawbacks are twofold. First, the use of privilege level hardware necessitates a privilege level 'switch' at every extension invocation, which, depending on extension size

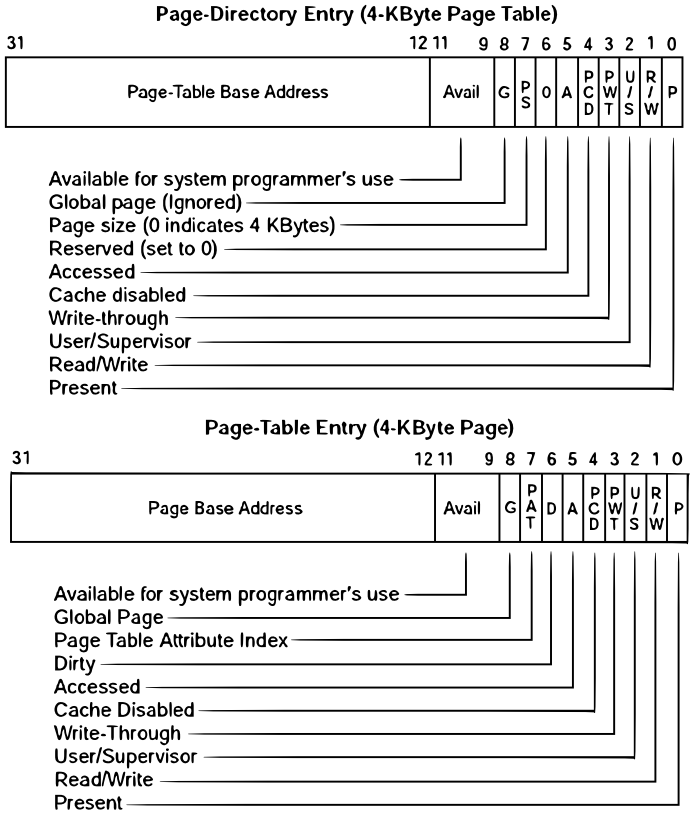

Figure 18: Page-Directory and Page-Table Entry formats for 4 KB pages. Source: *IA-32 Intel ® Architecture Software Developer's Manual, Volume 3: Architecture and Programming Manual.*

and hardware details, could be perceptible. Second, the reliance on hardware results in a strict and inflexible technique, in which the sets of allowed and disallowed instruction is fixed and immutable. For example, a kernel extension might have a valid need for control over interrupts and thus require the use of the `CLI` and `STI` IA-32 instructions to mask/unmask interrupts. If it is hardware code vetted through the use of sub-supervisor privilege levels, however, it would be unable to use those instruction, even though safe use could be arranged by means of non-maskable interrupt preemption.

3.1.2.3 Hardware Timers

Because software solutions to the timing isolation requirement are costly or impractical, hardware-based preemption solutions have been employed almost universally by systems ranging from consumer OS kernels to research extensibility frameworks. The approach used is to detect and preempt violations, *e.g.*, to preempt a process or extension that has failed to relinquish the CPU in time, rather than to attempt to eliminate the possibility of them occurring. The implementation of such preemptive approaches relies on the availability of hardware timers to provide the periodic detection and preemption points, as well as the means of accounting for the elapsed runtime. Typically, hardware timers in periodic mode interrupt the CPU at every expiration of a programmed period, thus giving the system a chance to decrement runtime quanta and, if needed, to preempt wayward software modules.

The IA-32 architecture provides a number of such hardware timers. More specifically they are the ubiquitous Intel 8254 Programmable Interval Timer (PIT), the Real-Time Clock (RTC), the Local Advanced Programmable Interrupt Controller (LAPIC), and the relatively recent High-Performance Event Timer (HPET).

The PIT and the RTC are the oldest timing sources, dating from the early 1980s, but they are also practically universally available. Both have a resolution of about $1ms$ and support periodic as well as aperiodic modes, though they are usually used in the former because of the costly I/O port accesses required to program them. The LAPIC timer was designed to synchronize multiple processors but suffered from poor resolution and silicon bugs and so it is rarely used in practice. Finally, the HPET was designed jointly by Intel

and Microsoft to supersede all others as the new IA-32 standard timer and to provide both periodic and aperiodic functionality, along with the fast memory-mapped programming and higher precision needed for modern multimedia applications.

Whichever the particular hardware timing source, it is used in essentially the same way to enable timing isolation. At entry into the code that is to be isolated, it is given a 'quantum', an initial time limit, and the timing source is programmed to initiate a periodic interruption that hands control back to the caller with a frequency which is a fraction of the quantum. At each interruption, the caller accounts for the elapsed time period by subtracting it from the callee's quantum. If the results in 0, then the quantum is exhausted and continuing the callee's execution would result in a timing violation. Instead, the interruption handler preempts it by forcing a return to its caller.

Timing isolation implemented in this way requires a hardware time source, but it offers a number of advantages over software alternatives. Its overhead consists of the initial timer setup and the brief periodic execution of the timer handler. Its only restriction is that the software module being isolated must not be allowed to meddle with the timer hardware or to block the delivery of its interrupt. This is typically achieved by some form of code vetting.

Because of the undecidability of the Halting problem [98], general formal methods for timing isolation do not exist. Software methods can be devised, which could instrument arbitrary programs to include a timing check in each of their basic blocks, or every loop and code path in the source, and thus ensure that they cannot be circumvented. The cost of such a scheme, however, is likely to be rather high, because basic blocks are rather short [36], resulting in a large 'timing' over 'useful' code ratio and significant runtime overhead. Exploiting call-graph information to minimize the that overhead by placing fewer checks in well-chosen locations, such as loops and alternative code paths, is possible, but it would complicate the instrumentation process significantly or limit the applicability of the technique to specific higher level languages.

3.2 Heterogeneous Hybrid

By their nature, extensible systems must balance the conflicting requirement of *safety* (*i.e.*, protection) with the *performance* goals pursued by their extensions. This dissertation is based on the premise that the trade-off employed by each single isolation scheme is not likely to meet the needs of all potential system extensions. To overcome such limitations, we propose a new approach to code isolation, a *hybrid* approach that aims to combine the best while avoiding the worst qualities of multiple extension techniques. As one concrete example, in the remainder of this chapter, we will describe the rationale, design, and implementation details for a software/hardware code isolation hybrid.

3.2.1 Metrics

Chapter 2 already explored one of the facets of the extensibility trade-off, namely, it analyzed the set of requirements necessary to guarantee the safety of an extensible system. But safety comes at a cost, and that cost is typically expressed either as some form of performance degradation, or as additional restrictions on the implementation or the usage of the system. These costs need to be measured carefully and balanced wisely against the costs of alternatives. We turn our attention now, to investigate an important question: "How do these costs present themselves and what are the relevant metrics that describe them?"

To answer that question, we will take a closer look at the usage model and the life cycle of an extension. Unlike regular software, once designed and implemented, extensions may still need an additional pre-processing step before they can be used to augment an application. The reason for this lies not only in the mechanics of dynamic linking of software components, but also, and more importantly, in the means of protecting the host application from its extensions. Such means vary widely depending on the particular isolation technique, from compiling an extension's source code with a trusted compiler, to binary rewriting its pre-compiled object code, to setting up hardware-restricted segments or pages for its code and data, *etc.* We term this additional pre-processing step the *build cost* of the extension to reflect its one-time preparatory nature.

Once set up, extensions are invoked as needed by their hosts, incurring two more traditional forms of runtime overhead, which we term their *latency* and *CPU burden*. From the viewpoint of the host software module, they reflect the pure cost of the control transfer into and out of the extension, and the pure overhead of the actual safe execution of extension code, respectively. While they are both also a function of the particular isolation technique, their impacts are rather different. The nature of the latency overhead is that it is incurred exactly once per invocation and is independent of the extension's size or instruction mix. Thus, latency is disproportionately more important for types of extensions that might require minimal reaction times, or that are small and/or frequently called upon. CPU burden, on the other hand, is a measure of the per instruction overhead of isolating the extension code. It can be sensitive to the extension code's instruction mix and proportional to its size, and thus it bears more importance for large and/or long running extensions.

Finally, all of the broad range of code isolation techniques employ different protection mechanisms and impose distinctive restrictions on important non-performance related parameters, such as extension size limits, types of code that a technique can be applied to, types of hardware that a technique can be used on, *etc.* We term this metric the *adequacy* of the code isolation technique. Even though it does not impact performance, adequacy can be of paramount significance, as it can rule out the possibility of employing a particular code isolation technique in a specific circumstance.

To summarize, our metrics for comparing code isolation techniques are as follows:

1. *Build Cost*: reflects load-time overheads,
2. *Latency*: reflects invocation overheads,
3. *CPU Burden*: reflects runtime overheads, and
4. *Adequacy*: reflects usability restrictions.

With respect to those metrics, Table 1 presents a compact contrast of how the state-of-the-art techniques compare to each other. Since our goal is to detect and to exploit the

Table 1: Comparison of isolation techniques' features.

	Build Cost	**Latency**	**CPU Burden**	**Adequacy**
Formal Methods	Proof Generation Complexity	None	None	Only Small Simple Codes
Type-Safe Languages	External Compiler	None	Low to Moderate	Language Lock, Recompilation
Software Fault Isolation	Binary Code Rewriting	None	Low to Moderate	Universal, but Platform Specific
Hardware Fault Isolation	None	Non-Trivial, HW Dependent	None	Universal, but Platform Specific

differentials that exist among homogeneous isolation techniques, we continue with a closer look at it.

The differences in the adequacy of a particular technique to a specific task are hard to quantify, as they are non-numerical and can depend on many divergent factors. The latter can range from the ability or inability of isolated code to run inside an interrupt handler, to whether the extension can be coded in some programming language, to whether legacy binary-only extensions can be handled. Build cost differences, on the other hand, are numerical and thus easy to quantify and compare. They are useful for distinguishing practical from impractical techniques, however, their optimization impact is relatively modest because of their one-time nature. For example, the build cost of proving the safety of a complex or large body of code like a complete device driver is currently beyond our grasp, whereas software techniques have been applied to the job. The cost differential between sandboxing a complex or large binary extension and invoking a type-safe language compiler for it, however, is not really decisive.

The most interesting trade-off exploitable in a composite hybrid isolation environment results from the cost differentials between techniques in the invocation latency and CPU burden cost categories. A key observation from Table 1 is that while software techniques incur minimal latency and a only a moderate CPU burden, hardware techniques have the exact opposite distribution of costs. This fact suggests possible potential for an improved hybrid solution, and merits a closer look.

The latency disparity is a direct result of the differences in the mechanics of control

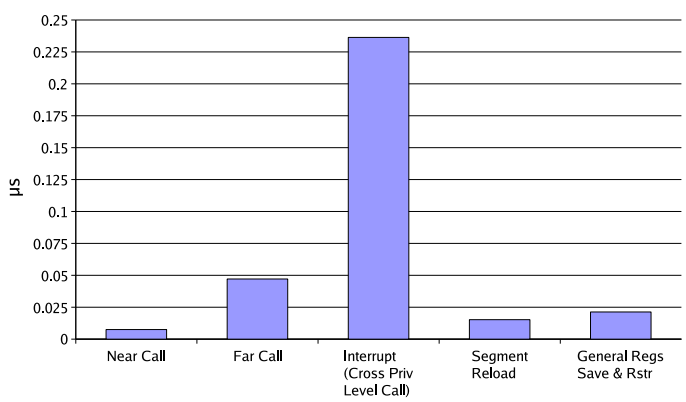

Figure 19: Basic costs of safe control transfers on an 866 MHz Pentium III. 'Near' calls are within the same segment, 'far' calls are to another segment, interrupts can be used to simultaneously cross segments and privilege levels, segment reload is the cost of reloading a data segment register, and finally the cost of general register saving and restoring.

transfers into and out of extensions for the two types of isolation techniques and their associated component costs shown in Figure 19. In general, all techniques require the saving and restoring of the CPU's general registers' state before and after the execution of foreign code. Even though such state is conventionally 'callee saved', extensions cannot be trusted to preserve it. The similarities, however, end there.

After state saving is completed, software fault isolation techniques can immediately transfer control into the extension, incurring only the additional cost of a 'near' function call. Hardware fault isolation techniques, however, require still more work to be done, in order to reconfigure the memory management unit of the CPU and fold memory isolation and/or code vetting checks into its normal operation.

Segmentation based HFI techniques must alter the set of referable memory segments to only those explicitly available to the extension, so they incur a number of data segment selector reloads and a 'far' branch to a different code segment at the same privilege level. If in addition they implement hardware code vetting, then the branch takes the form of a software interrupt, which can branch not only to a different segment, but also to a different privilege level.

Paging based HFI techniques are even costlier, as well as more difficult to quantify, because at extension invocation they must exchange the active page table and effect a TLB

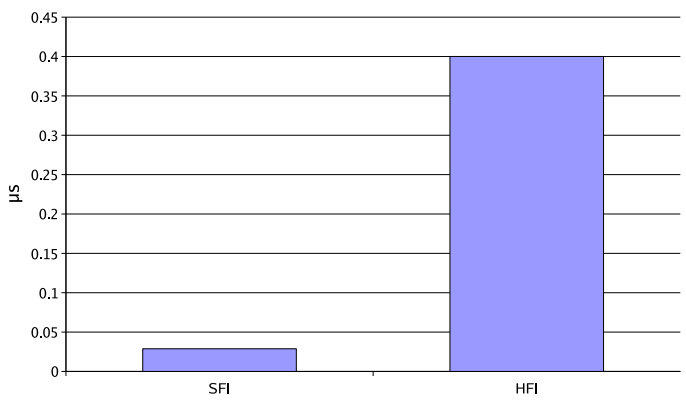

Figure 20: Aggregate costs of a safe control transfer on an 866 MHz Pentium III. The SFI cost includes saving and restoring general registers and a 'near' function call. The HFI cost includes saving and restoring general registers, reloading data segment selector registers upon entry and exit, and a branch to another segment and privilege level.

flush. The cost of the latter along with the subsequent costs of faulting-in the new page mappings, varies depending on the CPU architecture, *i.e.*, page table levels, as well as the data cache availability of the new mappings. Generally, paging is understood to be significantly more expensive than segmentation [38], since the overhead of reloading even a single TLB entry measures on the order of several thousand cycles [99].

To better illustrate the latency disparity, Figure 20 depicts it graphically by plotting the aggregate latency costs of memory isolation and code vetting implemented by purely software and purely hardware (segmentation) approaches where each bar reflects the sum total of their respective basic component costs.

Next, the reason for the CPU burden disparity lies in the way in which isolation techniques implement memory isolation. While immediate addressing memory accesses can be checked once at load time, the targets of indirect addressing memory accesses can only be known at run-time. This forces software fault isolation techniques to insert a number of additional checking instructions per each instruction that performs an indirect memory reference. The inserted code depends on the specifics of the SFI technique, but a general one could look similar to the ones shown in Listing 2 or Listing 3. Its performance impact is aggravated by the fact that indirect memory references typically comprise a substantial fraction of the total number of instructions in most programs. In fact, as evident from

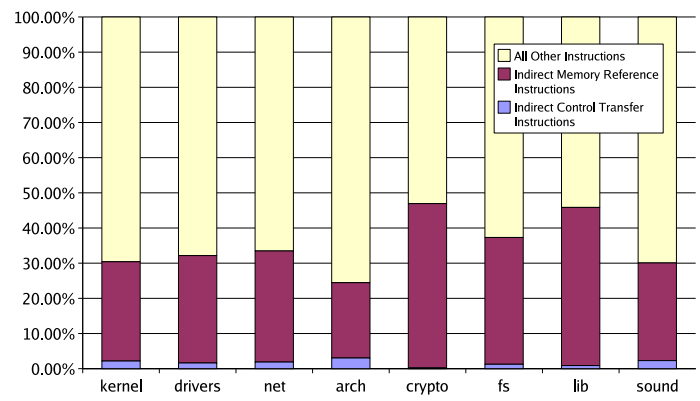

Figure 21: Typical extension instruction mix. Relative frequency of classes of instructions in typical systems extensions. The data was computed from a full set of the Linux 2.6.5 kernel's modules (977 modules total) by disassembling the compiled object files and counting the number of instructions falling in each category.

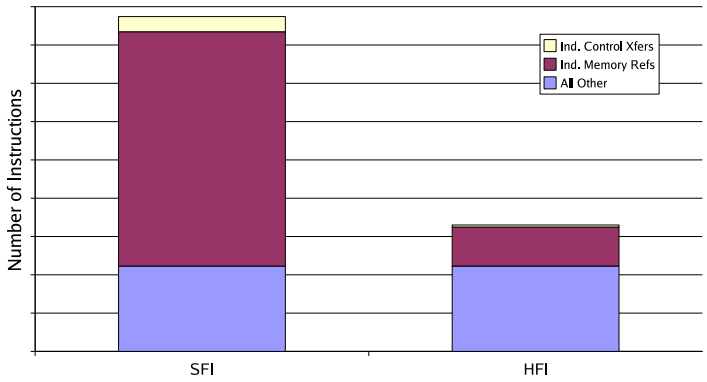

Figure 22: Code inflation of software fault isolation, projected on the basis of the examples in Listing 3 and Listing 10 and the average of the instruction mixes from Figure 21.

Figure 21, the relative frequency of indirect memory references in typical systems code varies around 25-30% of all instructions. The reason for that fact is that access to many fundamental programming language objects such as local variables, structures, unions, and arrays, has an inherently indirect nature and does not lend itself to static checking.

The dynamic checks, which are inserted into the binary at load time, can lead to a significant code inflation, and are the source of much of software fault isolation's CPU burden. For example, suppose that an extension's instruction mix contains 25% indirect memory references and suppose that, as in Listing 3, each one is isolated using up to 6

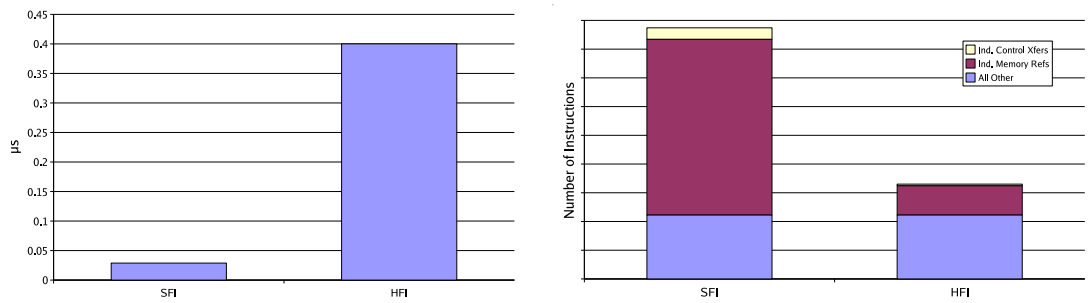

Figure 23: Inverse latency/CPU burden cost differentials that motivate our hybrid memory isolation and code vetting prototype. The left graph shows latency in μs, with the SFI case serving as a minimal baseline. The right graph shows CPU burden in terms of relative number of instructions (by instruction type), with the HFI case serving as a minimal baseline.

additional instructions. Under such plausible assumptions, the size of the pure isolation overhead could grow up to 1.5 times the extension's code size itself. To better display the magnitude of the effect, Figure 22 shows the relative code increase by instruction category in a reasonable systems instruction mix obtained by averaging all the categories in Figure 21 when the SFI techniques from Listing 3 and Listing 10 used to isolate it.

3.2.2 Cost/Benefit Differentials

The best opportunities for application of hybrid isolation techniques, therefore, arise in situations that exhibit all of the following characteristics:

1. *Existence:* There exist alternative techniques for implementing some of the isolation requirements,

2. *Feasibility:* There is a clear and consistent inverse cost/benefit differential between compatible pairs of alternative techniques covering a specific isolation requirement,

3. *Completeness:* A complete coverage of requirements is possible through some combination of alternative techniques.

Matching all of the above existence, feasibility, and completeness characteristics indicates potential benefits from a hybrid isolation solution.

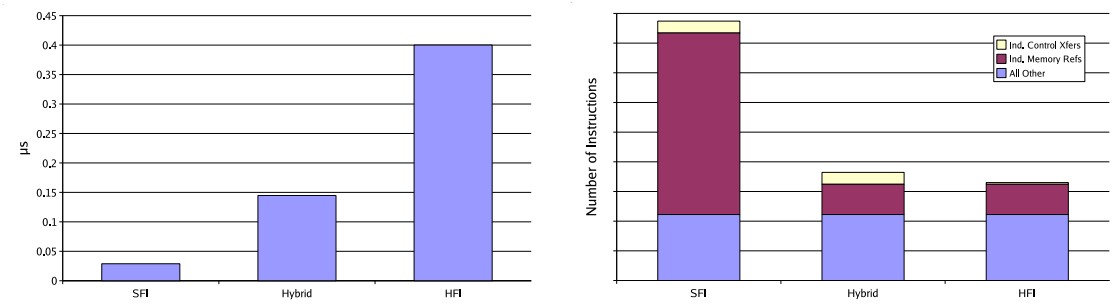

Figure 24: Comparison of latency and CPU burden overhead metrics among the hybrid prototype and software and hardware homogeneous techniques. The left graph shows latency in μs, with the SFI case serving as a minimal baseline. The right graph shows CPU burden in terms of relative number of instructions (by instruction type), with the HFI case serving as a minimal baseline.

For a concrete example, we turn to our proposed SFI/HFI prototype of hybrid memory isolation and code vetting. There, existence is satisfied by virtue of the fact that software and hardware fault isolation can each implement both memory isolation and code vetting properties. Feasibility is satisfied by the presence of an inverse latency/CPU burden cost differential, illustrated in Figure 23, and the compatibility of both techniques, *i.e.*, the lack of obstacles to applying *both* techniques to an extension at the same time. Finally, completeness is also satisfied, with each technique able to simultaneously satisfy more than one isolation requirement.

For our prototype, we assign timing and memory isolation to be covered by HFI, whereas code vetting and execution control are to be covered by SFI. The rationale for this assignment of implementation techniques to requirements is to maximize performance. Thus, timing isolation through hardware timers is chosen for its superiority over extensive instrumentation of every basic block. Memory isolation through segmentation is chosen for its lower CPU burden. And finally, code vetting and execution control through software binary rewriting are chosen for the reduced latency they offer. As expected, the resulting hybrid's performance characteristics, shown in Figure 24, are a balanced compromise between the extremes of its component techniques.

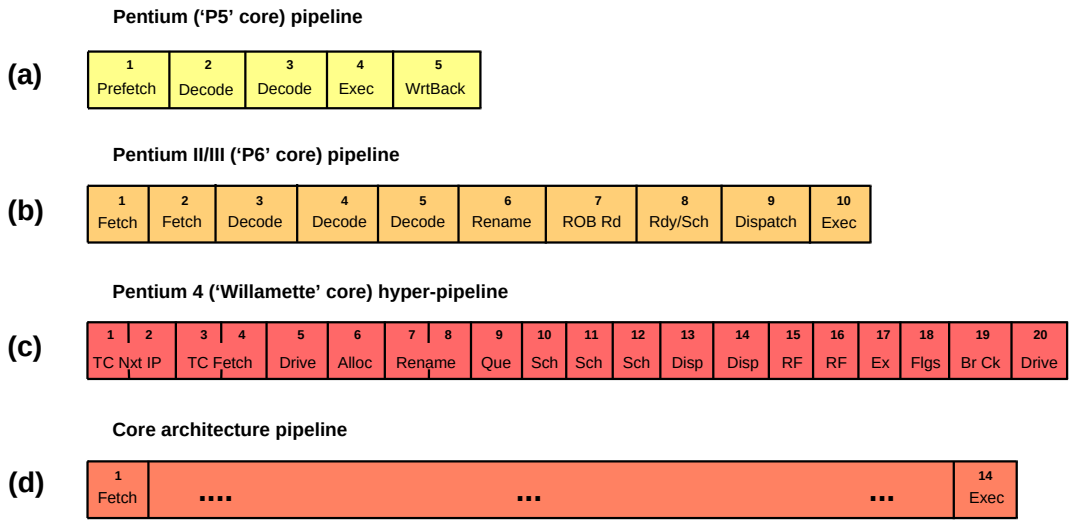

Figure 25: CPU pipeline depth evolution. (a) the 5 stages of the original Pentium processor's 'P5' core, (b) the 10 stages of the Pentium II and III processors' 'P6' core, (c) the much longer 20 stages of the Pentium 4 Willamette core design, later extended to 31 stages in the Prescott core design, (d) the latest 'Core' pipeline, a pronounced turn away from hyper-pipelines and extreme clock speeds, forced by power and heat dissipation problems.

3.2.3 Impact of CPU Micro-architecture

This work evolved out of our experience with the implementation of KPlugins, a purely hardware-based research facility for kernel extension, a complete description of which can be found in [29]. An important piece of critical feedback that we encountered during our experience developing KPlugins was the perceived high cost of control transfers into and out of plugins (for the remainder of this discussion we will use the terms 'plugin' and 'extension' interchangeably). Furthermore, Moore's Law and the processor architectural trends at the time pointed towards ever increasing CPU clock frequencies, accompanied by the corresponding deepening of super-scalar CPU pipelines. The Intel Pentium 4 family had been introduced with its new NetBurst ™ architecture including features such as hyper-pipelined technology and an execution trace cache, both of which made it radically different from the previous Pentium III's 'P6' architecture and aggravated the costs of plugin invocation.

The NetBurst cores increased the depth of the CPU pipeline from the original Pentium III's modest 10 stages (the 'P6' core), to Pentium 4's 20 (Willamette core) and even

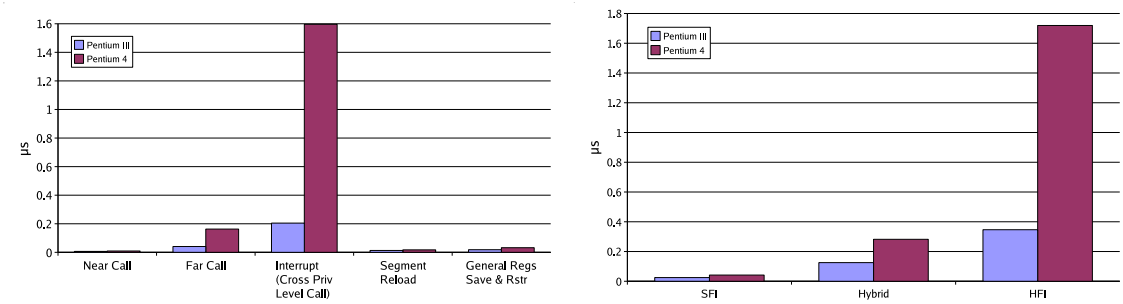

Figure 26: CPU micro-architectural effects on the overheads of code isolation. The left graph displays the disproportionate increase in basic HFI control transfer costs, whereas the right graph provides aggregate metrics. The 866 MHz Pentium III and 2.2 GHz Pentium 4 data used in these graphs were normalized to a reference clock speed of 1 GHz.

31 stages in the final Prescott core, shown in Figure 25. Compared to the previous generation, this effective *tripling* of pipeline depth decreased the amount of work that each stage must perform and allowed those Pentium 4 cores to achieve dramatic increases in operating clock frequency. Unfortunately, the increased pipeline depth also increased the number of stages that needed to be traced back and flushed when the CPU's branch predictor made a mistake.

Furthermore, NetBurst added the execution trace cache replacing the traditional L1 instruction cache. Instead of simply holding the CPU's original CISC instruction stream like its predecessor, the trace cache was designed to store its translation into internal RISC-like 'micro-operations', thus saving the significant effort of fetching and decoding them upon reuse.

These architectural changes, unfortunately, had a disproportionately adverse impact on control transfers that are difficult or impossible to predict, such as plugin entry and exit. The left graph in Figure 26 clearly shows the negative effect of the deeper pipeline on indirect addressing control transfers, particularly interrupts. In contrast, well-behaved register state saving and restoring code posts a net improvement, fully benefiting from the faster core clock speed, itself a result of the increased pipeline depth.

The exacerbated costs of hardware fault isolation on these hyper-pipelined architectures, shown in Figure 26, directly motivated this work, initially as an exploration of means to

alleviate the heightened plugin invocation latency. The rationale for our hybrid prototype was the observation, from Figure 26, of the unduly expensive nature of cross-privilege level control transfers compared to far function calls within the same privilege level.

The recent trend reversal away from over-clocked and hyper-pipelined CPU micro-architectures, such as Intel's 'Core', does not negate the benefits of hybrid designs. The benefits of hybrid designs generalize, and are not limited to the invocation latency *vs.* CPU burden trade-off used as a case study in this work.

3.2.4 Hybrid Prototype Technical Description

We continue with a closer look at the implementation details of our prototype of a hybrid safe extension facility for the Linux kernel. For the reasons already described above, our prototype was designed with a 50%/50% mix of hardware and software fault isolation techniques as follows:

1. *Timing Control* by preemption through periodic interruptions generated by a hardware timer,

2. *Memory Isolation* by memory segmentation at the host's privilege level using features of the CPU's MMU hardware,

3. *Code Vetting* by load time code sanitizing through a software disassembler, and

4. *Execution Control* by load time binary rewriting through a software disassembler/rewriter.

This combination allows it to combine the best characteristics afforded by the performance of MMU hardware assist for numerous and expensive timing and indirect memory reference checks, with the flexibility and low latency of software code vetting and indirect control transfers.

3.2.4.1 Timing Control

Our prototype's realization of timing control mirrors the usual operating system's scheme for preemption of threads and processes with some possible modifications. In fact, because

Listing 11: Sample plugin quantum decrementing and preemption code hooked into the periodic timer or NMI interrupt's exit path.

```
1           .
2           .
3           .
4           decl    SYMBOL_NAME(kp_quantum)  # decrement plugin quantum
5           jz      kp_quantum_expired       # quantum expired => terminate
6           .
7           .
8           .
```

our prototype is a Linux kernel extension facility, the timing control code is folded into the kernel's process preemption mechanism.

Upon boot-up, the Linux kernel programs some hardware timing source to deliver a periodic interruption to the kernel to serve as a well-defined point for process runtime accounting and preemption. On old systems the first programmable timer of the 8254 PIT is used, whereas on newer systems programmable timers in either the LAPIC, or the HPET are used. Irrespective of the actual source, the interrupt is delivered to the same software handler and provokes the same sequence of events including the decrementing of the quantum for the currently running process and its de-scheduling if the latter has expired.

At the point of interruption, the CPU is normally running in either unprivileged user mode, executing the current process, or in privileged kernel mode, on behalf of the current process. With the addition of our kernel extension facility, a new alternative is created, namely that the CPU could be executing a kernel extension on behalf of a kernel service.

Accordingly, our prototype modifies the usual Linux timer interrupt handler, as well as all other interrupt and exception exit paths to add a check for this eventuality. When a plugin context interruption is detected, similarly to process runtime accounting, plugin runtime accounting is performed by decrementing a separate plugin quantum and checking for its expiration on the interruption exit path as shown in the assembly in Listing 11. Unlike process preemption, however, plugin quantum expiration results in the termination of the offending plugin. The reason for this is that in our simple prototype's design, plugins are not schedulable entities. Rather, they are expected to run to completion in their allotted time.

The newly introduced plugin context processing is an addition to the usual Linux kernel's interrupt and exception routines. It is complementary and does not interfere with traditional thread and process quantum decrementing and preemption.

A possible modification to the described scheme is to re-program the CPU's LAPIC to deliver a periodic NMI, or Non-Maskable Interrupt, and then use it to drive plugin quantum accounting and preemption. The benefits of such a modification are that it would be operable even while the CPU's usual interrupts are disabled, thus permitting us to allow otherwise unsafe interrupt control instructions in untrusted plugins. Our simple prototype does not implement this modification because of the complexity of re-programming the LAPIC hardware. Despite its obvious usefulness in a production system, the implementation of such a capability is not essential in a proof-of-concept prototype like ours.

3.2.4.2 Memory Isolation

To cover the memory isolation requirement, our prototype employs a hardware segmentation based technique. Plugins are placed in memory described through a number of specially created segments that are different from the kernel's, but whose memory is a physically a subset of its. Thus, the kernel has full access to plugin memory through its all-encompassing segment descriptors, whereas the plugins themselves live in the smaller, restricted world. The plugin code and data segments cover disjoint memory regions. The former permits reading and execution and the latter reading and writing of its memory.

Additionally, we have provisions for the creation, on a need-to-know basis, of an extra plugin data segment that maps some subset of the kernel proper's data space read-only. It can be useful for providing immutable plugin access to kernel structures such as received network packets or file system buffers.

Every time the kernel invokes a plugin extension, it reloads the CPU's segment selector registers (`CS`, `SS`, `DS`, `ES`, `FS`, and `GS`) to point to the plugin segment descriptors. The code vetting and execution control properties ensure that plugins cannot reload or alter the segment parameters during runtime and the MMU guarantees that every subsequent memory reference conforms to the types and ranges of the described memory segments.

Upon completion, plugins return to a well-defined entry point into the kernel, which reloads all segment selectors to their usual kernel values before proceeding.

The scheme is similar, but not identical, to fully hardware fault isolated systems such as Palladium [16, 100], the L4 microkernel 'small spaces' [55, 99], or the original Kernel Plugins [29]. The important distinguishing factor is that those systems employ both the CPU's segmentation and privilege level hardware by assigning plugin segments to a separate privilege level from the host kernel.

The assignment is encoded in each segment's DPL or Descriptor Privilege Level bits, shown in Figure 17, combined with the RPL, or Requested Privilege Level in the segment selector register's least significant 2 bits, and the processor's CPL, or Current Privilege Level, defined as the RPL of the current code selector register. Typically all three are the same, though more complicated relationships are possible but outside of the scope of this narrative. For a complete discussion the reader is referred to Chapter 4.5 in [49].

In this way, those techniques attain not only memory isolation, but also a rigid form of code vetting, as the lower privilege levels limit the allowable instruction set. Our prototype differs by explicitly assigning plugin segments to the kernel's privilege level. This enables us to:

1. Avoid costly CPU burden and maintain native execution speed while isolating indirect memory references, which make up roughly 30% of typical systems code (see Figure 21),

2. Enable arbitrarily flexible code vetting, by shifting coverage of that isolation requirement to more fluid software methods, and

3. Achieve shorter invocation latency, as cross-privilege level control transfers are significantly costlier than same-privilege level ones (see Figure 19).

3.2.4.3 Code Vetting

In order to obtain maximum flexibility with respect to the types and formats of extensions that our prototype can handle, we employ a binary rewriting approach that operates on raw

machine code and thus applies directly and easily to arbitrary languages and compilers. At plugin load time the rewriter performs code vetting by decoding and sanitizing the plugin.

Depending on its requested mode of operation, it can either verify the absence of dangerous systems instructions and reject plugins failing the check, or accept such plugins after neutralizing the dangerous instructions by converting them to null operations, *i.e.* to a sequence of `NOP` instructions on the IA-32.

The prototype implementation of our binary rewriter, termed 'librewrite', is based on the 'libopcodes' library, which is a part of the 'binutils' package supporting the GNU Compiler Collection [27]. The library provides the cross-platform disassembler functionality that is used in 'gdb' – the GNU project debugger [26].

The original library's typical usage mode consists of two phases: initialization and disassembly. Prior to use, the initialization routine must be invoked to set up the target architecture, the disassembler format requested, and the region of memory holding the machine code of interest. The disassembling routine can then be invoked repeatedly at arbitrary addresses within the buffer region, and it will decode and return the assembly for the instruction aligned at that address along with its byte length, *i.e.* the buffer memory consumed. One can then restart the process by advancing the disassembling pointer by that length, and re-invoking the disassembler.

The library is driven by a structured tabular description of the target architecture's assembly language. Our prototype extends the basic descriptive structure with a number of additional fields as follows:

1. *sys_insn*: A data flag that specifies the category of the described instruction. It could classify it as either a safe non-systems instruction, *e.g.*, `ADD`, `PUSH`, *etc.*, or as an unsafe systems instruction, *e.g.*, `INT`, `LDS`, *etc.* The latter form the class of code which is not permissible in plugins by the security policy embodied by the table. Depending on the mode in which it has been invoked, the presence of systems instructions in its input causes the sanitizer to either reject the plugin outright, or to neutralize all unsafe instructions in its output.

2. *need_scheck*: A data flag that marks the instruction as requiring static checking. Typically, these are safe instructions that operate on or branch to directly addressed memory, the address of which can be verified at load time. In these cases no further runtime verification is required because of our prototype's immutable code invariant. Examples include directly addressed forms of instructions such as `MOV`, `PUSH`, `JMP`, *etc.*

3. *need_dcheck*: A data flag that marks the instruction as one that cannot be checked statically and thus requires dynamic checking. Typically, these are safe instructions that operate on or branch to indirectly addressed memory. The effective address of their targets cannot be computed and verified before their actual execution and thus they require that additional checking code performing the verification be spliced into the original instruction stream before them. Examples include indirect memory addressed forms, *e.g.*, `0x42(%esi,%eax,4)`, of instructions such as `MOV`, `PUSH`, `JMP`, *etc.*

4. *pre_xform*: A function pointer that points to a procedure to be invoked prior to processing the described instruction. The transformation typically emits checking code relevant for that class of instructions, injecting it immediately before the trigger's actual encounter in the output stream.

5. *insn_xform*: A function pointer that points to a procedure to be invoked to transform the actual instruction bytes and pass them on into the result buffer with or without any required modifications.

6. *post_xform*: A function pointer that points to a procedure to be invoked after processing the described instruction. The transformation typically emits checking code relevant for that class of instructions, injecting it immediately after the trigger's actual encounter in the output stream.

The resulting structure is no longer a passive representation of the structure of the architecture's machine language. Along with the relevant processing functions, it is now an active object that embodies a particular code vetting policy. Combined with our modified

disassembling routine, it can be used to apply, both statically and dynamically, the isolation policy to a piece of machine code produced by any compiler from any high-, or even low-level, language.

Similarly to the libopcodes disassembler, our binary rewriter is driven by the augmented table. However, in contrast to the former, it is designed not as an inspection tool, but as a filter or a mapping function that transforms the input machine code into a new, functionally equivalent machine code guaranteed to conform to the implicitly encoded security policy.

Librewrite's design is powerful in its capacity to accommodate, encode, and enforce arbitrary machine instruction-level security policies through the combination of its classification markers and transformation actions. Its implementation is also flexible, allowing it to compile both as a user-space library and as a kernel module, in order to support a wide spectrum of projects. Furthermore, the computational complexity of librewrite, like that of its precursor, is linear thanks to their simple table-driven structure.

In the larger context of our hybrid code isolation prototype, however, librewrite is used for code vetting and execution control only. While the generality of the rewriter is not needed for the former where a simple load-time sanitizing would suffice, the latter makes full use of its powerful transformation hooks.

3.2.4.4 Execution Control

As already described in Section 3.1.1.3, our improvement proposal for SFI execution control prescribes that: (1) we build a bitmap recording the boundaries of all well-aligned instructions in the output plugin and that we instrument all indirect control transfer instructions so that during runtime the effective addresses of their targets are validated through that bitmap. We implement this needed functionality by taking advantage of the rewriting hooks of librewrite.

The bitmap is built by first allocating and pre-zeroing sufficient memory space for it and then gradually setting only the bits corresponding to the beginnings of all well-aligned instruction during the rewriting process.

Target address validation code is injected from the pre/insn/post transformation hooks.

First the effective target is computed using the original instruction's indirect addressing mode. The absolute address is then converted to a relative offset within the plugin, and finally the bitmap lookup is performed at that relative index. Direct addressed control transfers are also verified by bitmap lookups, though only statically at load time, so they do not result in the injection of verification code.

This approach leads to a couple of interesting problems that merit mentioning, which are (1) how to deal with direct addressed forward references during the rewriting when the bitmap is still incomplete, and (2) how to handle the internal shifts in the output binary resulting from the injection of the extraneous validation instructions.

Our prototype solves the first problem by implementing lazy verification of direct addressed forward references. When they are encountered, we record their target address in a sorted linear list along with the referencing instruction address. A number of these records to the same target are possible when multiple control transfer instructions branch to the same target, *e.g.*, the 'breaks' in a C switch statement. As plugin code is processed, if the current rewriting pointer passes the target address at the front of the forward reference list we handle all the newly eligible records by back-checking their referencing instructions before proceeding further.

Our solution to the second problem is to maintain a second structure recording the amount and location of code 'inflation' caused by the injection of verification code. The inflators structure is implemented as a vector array of records sorted by inflator address. The linear nature of the rewriting process ensures that the vector of inflators is maintained naturally and with low overhead. In addition, to minimize the cost of inflation lookups, each record reflects the *cumulative* code inflation starting at its address and until the next record's address. Thus, the monotonically increasing nature of the structure provides a useful sanity checking invariant. Coupled with the forward reference list, the table of inflators provides a convenient way of re-computing and back-patching forward reference targets of control transfers at the same time when those targets are lazily validated.

At the completion of the rewriting process the final bitmap reflects all well-aligned instruction boundaries. All directly addressed control transfers had been checked against it,

and all indirectly addressed ones have been prefixed with runtime validation code. Finally, because the initial entry point into the plugin is determined by its caller, it follows by induction that execution control has been established over the entire plugin.

3.3 Other Hybrids

In conclusion, we make no claims of uniqueness or optimality for the particular hybrid prototype described above. Instead, it is intended as an example of the general idea of hybrid code isolation advanced in this thesis, with our implementation prototype as a simple proof-of-concept. Many other hybrids are indeed possible, *e.g.*, combining different homogeneous donor techniques, a different balance of individual functional contributions, *etc.* The specific hybrid technique incarnation described in this chapter was selected for the availability of hardware, software, documentation and, last but not least, for practical implementation reasons.

CHAPTER IV

EXPERIMENTAL EVALUATION

The hybrid isolation approach provides a powerful trade-off mechanism for adjusting or optimizing parameters of code isolation schemes. This chapter shows how that mechanism can be applied, in practice, to a number of different applications. It also evaluates the advantages of such hybrid designs by more carefully quantifying the cost differentials involved and the benefits that can be extracted.

The evaluation will consist of a two-part study of micro-benchmarks and realistic macro-applications. Micro-benchmarks are a useful way of focusing attention on specific individual parameters of the isolation scheme, in a fashion that controls other variables. Macro-applications, on the other hand, present a complex environment where many individual effects are intertwined and thus, are not easy to distinguish. Despite that, such complex environments better reflect the actual real-world scenarios in which hybrid isolation schemes are ultimately to be used. The ability to demonstrate a certain net benefit in realistic conditions serves to provide validation to the practical usefulness of hybrids.

The chapter starts with a description of the methodology used in setting up and executing experiments, including a discussion of the measurement techniques employed. Then, we characterize the experimental platform and present the micro-benchmark and macro-applications' results. The chapter concludes with a brief analysis and interpretation of the data.

4.1 Methodology

The experimental methodology used in this evaluation is aimed at achieving a pure body of performance data to work with by eliminating as many extraneous effects as reasonably possible. We will strive to employ the most precise timing methods available and to reject or cancel out non-deterministic and/or non-linear data pollution effects resulting from both hardware and software sources that can be as varied as out-of-order instruction execution,

memory caching, hardware interrupts, OS scheduling, *etc.*

Our approach is two-prong. First, we pursue an implementation that rejects as many of the underlying sources of data pollution as possible by design. Second, in order to cancel out the remaining factors, we will employ statistical measures of central tendency such as medians, means, and variances, in order to filter outliers from the data sets. In the next two sub-sections we elaborate on the details and implementation issues of those two approaches.

4.1.1 Timing

Because in this evaluation we will be concerned with efficiency measures such as latency and throughput, we will naturally need to measure very small time periods with high degree of accuracy, as well as to count the number of events occurring per units of time. Time, then, emerges as an important parameter that we need to be able to measure with precision.

Timing Mechanisms

The IA-32 platform provides a number of hardware assets for time measurement such as the PIT, the LAPIC, or the HPET. All of those are capable of periodic as well as one-shot modes, but are more suitable for asynchronous notification of the expiration of a preset time period, rather than for use as a stopwatch. Still, when programmed in periodic mode they can be used as the latter by counting the number of their ticks taken by the measured activity. However, those assets are typically implemented as separate programmable chips, and they require non-trivial effort to set up, initiate, and conclude a measurement [20]. Those setup costs can involve slow input/output instructions, *e.g.* as with the PIT, and unpredictable data bus transaction latencies, *e.g.* as with the LAPIC or HPET, and can distort the period being measured significantly, especially when that period is close to or smaller than the hardware timer's period.

For these reasons, we choose to employ a different timing mechanism available in all Pentium class and later IA-32 chips. The method relies on the use of the new `RDTSC` instruction, whose name is an abbreviation of *'ReaD Time-Stamp Counter'*.

All Pentium and later chips from Intel Corp. implement an internal 64-bit time-stamp

counter register in their CPU cores, which is cleared at every processor reset and is incremented by exactly 1 with every core clock cycle. With current power and heat budgets capping the top processor clock speed in the 3 to 4 GHz range, the counter is large enough to accommodate continuous counting without overflow for approximately 146 to 194 years, depending on actual core clock frequency, which is sufficient to cover any reasonable uptime.

The `RDTSC` instruction's effect is to read the momentary value of the hidden internal 64-bit counter register and to return it in the user-accessible pair of `EDX:EAX` 32-bit registers [48]. Because the `RDTSC` instruction is usually freely accessible in every CPU mode, it allows for the easy, convenient, and extremely low-overhead measurement of time periods with very high precision. The typical CPU core's operating frequency today ranges anywhere between 1 and 4 GHz, resulting in theoretical precision of 1/4 ns. Unfortunately, the practical limits of `RDTSC`-based timing are less precise because of complications arising from the out-of-order execution engines that power almost all modern super-scalar processors, including the ones offered by Intel Corp. and AMD.

Out-of-order execution (OoO) is a processor implementation technique that allows the recapture and use of processor cycles that would otherwise have been wasted because of delays in pulling data dependencies from slower peripherals, *e.g.*, main memory, or because of shortages of internal CPU resources, *e.g.*, renaming registers. OoO execution allows for the retirement of instructions from the CPU core before independent preceding instructions, in the sense of the linear program order, have retired. This optimization increases the Instructions Per Clock (IPC) metric, a key measure of CPU throughput.

In the case of `RDTSC`, the OoO nature of IA-32 processors poses a problem, in that it reduces the practically attainable timing precision from its theoretical limit. Since the `RDTSC` instruction has no dependencies, it is often able to overtake and commit prior to others that appear before it in the linear program order but which have stalled waiting for data dependencies, for the availability of ALU units, *etc.*

In essence, this allows the setting of the start and/or the end of a timing period to occur non-deterministically earlier than the linear program order would suggest, *e.g.*, a scenario shown in Listing 12, or that subsequent instructions be executed before the time-stamp

Listing 12: A simple example of likely out-of-order timing instruction execution. The fact that the memory cells holding the multiplication operands are missing from the data cache would stall the `MUL` instruction until they can be fetched from memory. Because there are no dependencies between the `MUL` and the `RDTSC`, the latter can overtake the former and commit out-of-order, presuming that enough internal resources are available to rename the temporary result register from the multiplication that gets overwritten by `RDTSC`.

```
1 movb   UNCACHED_MEM_A,%al   # load first mul operand
2 mulb   UNCACHED_MEM_B       # multiply by second operand
3 movw   %ax,RESULT           # save result
4 rdtsc                       # take a time-stamp
```

counter reading has committed. While out-of-order execution would have little impact on the timing of relatively long-running events, such as thread time slices or long running subroutines, it can introduce significant errors into the measurement of short events, such as individual instructions or even small high-level language procedures.

The solution to this problem is the *serialization* of the instruction stream just prior to the acquisition of each time-stamp. Serialization refers to the explicit imposition of a strict serial ordering between the in-flight instructions already issued before the serializing instruction and the ones following it. Serialization also imposes an ordering on memory transactions issued by preceding instructions, so it can be quite expensive. In essence, serialization might induce a 'bubble' in the execution pipeline of super-scalar CPUs, by forcing some wasted wait cycles until all in-flight instruction and their memory side effects have safely committed before continuing. Despite that fact, serialization is vital for the achievement of precise timing results when measuring extremely short periods.

The proper way to enforce serialization on the IA-32 platform is by issuing a serializing instruction just before the `RDTSC` timing one. `CPUID` is a popular unprivileged serialization instruction typically used for that purpose because it is short, simple, and has no dependencies. It can go through the CPU pipeline with minimal conflicts, retire quickly, and cause the smallest perturbation on the timing task at hand. Example time-stamping code utilizing `CPUID` serialization is shown in Listing 13.

These additions to the simple act of time-stamping necessarily lead to a dilution of the attainable timing precision. The practical precision limit, however, can be determined by

Listing 13: Properly serialized collection of a time-stamp. `CPUID` serializes the instruction stream, guaranteeing that all instructions that have already issued and their effects will commit before the following instructions. `RDTSC` acquires a time-stamp into the `EDX:EAX` register pair, and the final two instructions save it into memory.

```
1 cpuid                          # serialize instruction stream
2 rdtsc                          # take a time-stamp
3 movl    %edx, HIGH_BITS        # save the high 32-bits of time-stamp
4 movl    %eax, LOW_BITS         # save the low 32-bits of time-stamp
```

observing the spacing between two successive back-to-back serialized time-stamps readings. For our experimental platform, described in detail in Section 4.2, that practical limit is 168 cycles, computed by the code in Listing 14.

Timing Disturbances

Another complication when attempting precise event timing measurements can arise from the interference that simultaneous, but unrelated events, can have on the one under measurement. The typical source of such interferences are the interrupts generated by hardware devices.

Interrupts are an asynchronous notification mechanism used widely from system timers to network interface cards, fixed disks, *etc.* They are beneficial as they eliminate the need for the CPU to perform constant polling of the status of the devices. Their drawback is that their inherently asynchronous nature causes them to occur at generally unpredictable times and to cause interruptions in the normal flow of other program code.

When carrying out timing experiments, those interruptions can alter the performance of the subject code not only by inducing secondary effects such as data and code cache pollution [66], but also by directly adding their handlers' execution time to the timed event.

Most OSes today run at least a part of the interrupt processing load in the context of the application which happens to be executing at the point of interruption. The interrupt time is unfairly billed to the application and should the interruption occur within the span of a event that is being timed, they inflate and corrupt that measurement.

For example, in Microsoft's Windows OS 'Deferred Procedure Calls', or DPCs, handle the bulk of the interrupt processing load and consume the processing time of the currently running thread [63]. The DPC's equivalent in the Linux kernel are the so-called 'soft IRQs',

Listing 14: This C code is used to compute the cost of serialized `RDTSC` time-stamp collection. Back-to-back serialized time-stamps are taken, their difference is computed and aggregated. After a preset number of repetitions the average is obtained by dividing the sum by the number of repetitions. Note 1: the first difference sample is disregarded to avoid cold data caches. Note 2: the code snippet uses the `gcc` compiler's inline assembly extensions.

```
#include <stdio.h>

#define REPT_SHIFT  (10)
#define REPT        (1 << REPT_SHIFT)

unsigned long long ts1;
unsigned long long ts2;
unsigned long long sum = 0ULL;

int main()
{
    register int i;

    for(i=0; i<=REPT; i++) {
        __asm__ __volatile__(
            /* Save registers */
            "pusha\n\t"
            /* Serialize, time-stamp, save to memory */
            "cpuid\n\t"
            "rdtsc\n\t"
            "movl %%eax,ts1\n\t"
            "movl %%edx,4+ts1\n\t"
            /* Serialize, time-stamp, save to memory */
            "cpuid\n\t"
            "rdtsc\n\t"
            "movl %%eax,ts2\n\t"
            "movl %%edx,4+ts2\n\t"
            /* Restore registers */
            "popa\n\t"
            : : : "memory");

        /* Compute time-stamp difference */
        ts2 -= ts1;

        /* Discard first sample (cold data cache) */
        if(i) { sum += ts2; }
    }

    /* Compute & print average */
    sum >>= REPT_SHIFT;
    printf("avg = %llu\n", sum);

    return 0;
}
```

Listing 15: Example codes which (a) disable and (b) re-enable hardware interrupts on the local processor.

```
# (a)
        pushf
        cli

# (b)
        popf
```

and in recent versions of the kernel, soft IRQs have been migrating into special separate kernel threads per CPU. While this improves on the former problem, it may not alleviate the latter one, as that is dependent on the priorities of the system scheduler, and in all cases, *some* 'hard' interrupt processing occurs within interrupt handlers and is incorrectly billed to tasks.

There are two possible approaches for correcting such timing disturbances. The first is to take measures to prevent their occurrence, and the second is to adjust the measurements taken in order to compensate for or to cancel out the disturbances post-factum.

The former approach is attractive because of its simplicity and ease of implementation. In essence, all that is required is to disable hardware interrupts before initiating a measurement, perform the measurement, and re-enable hardware interrupts after it. The code to do that is simple, consisting of just a few instructions as shown in Listing 15. The simplicity is misleading, though, as there are significant disadvantages to this approach. Disabling hardware interrupts for long periods of time can not only impact the performance of multimedia applications, as well as OS time-keeping and scheduling, but also impair the correct operation of hardware devices. These drawbacks can be mitigated somewhat by either only measuring short events, or by performing the measurements on a multi-processor system where hardware interrupts are routed by a smart interrupt controller that is able to forward them for processing to a CPU that is able to take them.

The latter approach is more complex as it requires additional instrumentation to track the occurrence of hardware interrupts during event measurements and to record their CPU usage. It also requires a post-processing step to subtract the interrupt processing from the event timing samples. Furthermore, as with any other benchmarking instrumentation, one

always needs to be careful with implementation so as not to impact the event being studied. The additional variables this approach must track can increase that risk.

For the purposes of this dissertation we have chosen to implement the former approach as its advantages outweigh its drawbacks in this case. The events being measured are short enough not to interfere with the correct operation of hardware and the simultaneous performance of multimedia applications and OS time-keeping and scheduling are not critical for the relatively brief benchmarking periods.

4.1.2 Statistics

Irregularities

Even when all efforts are taken to avoid timing disturbances, measurement samples are subject to unavoidable regularities arising from the complex nature of modern computing platforms. The ever growing performance gap between processors and memory systems and the increasing data appetite of larger super-scalar CPUs have necessitated the introduction of fast *cache* memories at various points at the CPU/memory interface to buffer recently used data and instructions and exploit the locality of reference exhibited by most programs.

While caches have been remarkably successful in compensating for the computation/storage performance mismatch, they have also introduced significant irregularity in the computing architecture by making memory accesses unpredictably non-uniform. The formula expressing the average time for memory accesses in the presence of caching is:

$$T_{avg} = P \times T_{cache} + (1 - P) \times T_{DRAM}$$

where T_{cache} is the cache access time, T_{DRAM} is the much larger main memory access time, and P is the *probability* of a cache hit. To complicate matters further, cache misses can be predictable, *e.g.*, *compulsory* misses incurred on the first reference to a datum, or unpredictable, *e.g.*, the *capacity* misses due to the finite and insufficient size of the cache itself, or the *conflict* misses caused by imperfect cache mapping functions and/or replacement policies.

It is clear then, that the typical memory access time of a modern computing architecture is a probabilistic variable, dependent on a number of varied parameters, such as the cache hardware's size and associativity, the program's data access pattern, the program's particular input data vector, and even events external to the program itself like interrupts and other threads executing in parallel.

Thus, the probabilistic nature of the caches' performance and their increasing use for not only data (D-caches), but also instruction fetching and decoding (I-caches), page-table mappings (TLBs), virtualization control structures (VMCS-caches[11]), *etc.*, can induce complex consistency and/or replication irregularities into benchmarking experiments.

While such irregularities are unavoidable, their effects can be mitigated by careful experiment design and the use of statistics. In this dissertation, we employ the following techniques to alleviate irregularities and improve benchmark consistency:

1. Disable hardware interrupts to reduce interrupt handler I&D-cache pollution during benchmarking.

2. Repeat benchmark runs and discard the initial values to eliminate compulsory, or cold-cache, misses.

3. Employ measures of statistical central tendency and variability, such as medians, means, and variances, in order to more accurately estimate actual performance behavior.

4.2 Experimental Platform

Irrespective of measurement precision, performance results are relative and a function of the actual platform, in both the hardware and software sense, where they were obtained. Hence, in this section we will describe, as completely as possible, the details of our experimental platform.

Hardware

Most of the results in this dissertation are obtained on a personal computer with the

Table 2: Experimental platform's hardware specifications.

Feature	Part & Model
CPU	2 x Intel® Pentium® III @ 866 MHz (16 KB 4-way I1&D1 caches, 256 KB 8-way unified L2 cache)
Motherboard	ASUS CUV4X-D (Dual Socket370)
North Bridge System Chipset	VIA® 82C694XDP (133/100 MHz FSB and 4X/2X AGP)
South Bridge System Chipset	VIA® 82C686B (UltraDMA/100/66)
IDE Ports	2 x UltraDMA/100 Bus Master IDE Ports
Expansion Slots	1 x AGP PRO/4X and 5 x PCI
I/O Ports	1 x PS/2 keyboard, 1 x PS/2 mouse 4 x USB v1.1 2 x serial, 1 x parallel 1 x game, 1 x audio
Memory	768 MB of PC100 SDRAM
Video	NVIDIA GeForce FX 5200 with 256 MB of Video RAM
Video Capture Device	Creative Labs Video Blaster WebCam 3 USB (Model No. CT6840, CMOS Sensor OmniVision OV7620)
Network	Intel EtherExpress PRO100 PCI adapter
Disks	1 x 20 GB Western Digital WD205AA (UDMA4 mode) 1 x 30 GB Western Digital WD300BB-00AUA1 (UDMA5 mode)

specifications listed in Table 2. Because of an unfortunate hardware failure (a CPU malfunction), however, some measurements had to be performed on a slightly altered system, where only one of the CPUs was present.

While the presence of a second CPU has a beneficial effect on the overall performance of the system, it does not impact the fundamental trade-off on which hybrid code isolation is based, *i.e.*, the cost differentials between software and hardware code isolation, or between control transfers to the same or to a different privilege level. Nevertheless, those results which have been obtained on a uniprocessor configuration are clearly marked.

The choice of an experimental platform based on the Pentium III, a processor two generations removed from the current state-of-the-art, might seem like an odd one, yet, it was motivated by solid technical reasons.

The next processor generation, the Pentium 4, is based on the vastly different NetBurst™ micro-architecture. NetBurst was characterized by a radical departure in micro-architecture

design, most significantly because of its hyper-pipelined nature. Its various incarnations increased the CPU pipeline depth from Pentium II and III's 10 stages, to 20 in the Willamette core, and even 31 stages in the final Prescott core. The dramatic increase in pipeline depth reduced the workload of each stage and permitted a similarly dramatic decrease in the core's basic clock cycle, thus paving the way for achieving much higher operating frequencies. NetBurst cores were aimed towards scaling operating frequencies up, eventually reaching up to 10 GHz.

Unfortunately, once NetBurst based processors started production, they were unable to deliver the promised performance, beset by a combination of architectural and engineering problems. Their hyper-pipeline proved difficult to utilize efficiently. Its higher operational frequency and larger depth exaggerated the processor/memory performance gap and aggravated the already high costs of pipeline dependency stalls and flushes due to mispredictions. At the same time, the high operating frequency resulted in non-linear increases in power and heat dissipation.

Realizing the magnitude and complexity of the problems, Intel decided to abandon NetBurst and return to older designs and refocus on IPC (Instructions Per Clock), power consumption, and heat production. The current state-of-the-art Core™ and Core 2™ micro-architectures bear a lot of similarity to the Pentium M, itself an evolution of the Pentium III, and reflect micro-architectural trends for the foreseeable future.

For these reasons, lacking a current Core 2 based platform to experiment on, we opted for the next best thing and chose a Pentium III based platform. Although it is only an approximation of the current state-of-the-art, it is relevant with respect to future micro-architectural trends.

Software

The software configuration of our experimental platform is listed in Table 3.

The choice of operating system kernel was restricted by the fact that our hardware fault isolation implementation was based on the 2.4.24 version of the Linux kernel. Porting our prototype to the current 2.6 series of kernels would have required a non-trivial reimplementation effort with little research value.

Table 3: Experimental platform's software specifications.

Feature	Release
OS Distribution	Fedora Core 1 (Yarrow) with all updates applied
Kernel	Linux 2.4.24 with research patches
Web Server	Apache 2.0.47
Video Capture Driver	USB OV511+ (drivers/media/video/ov511.o)

The particular choice of kernel led us to the selection of Fedora Core 1, FC1 for short, for the distribution, as it is the most recent Linux distribution that is compatible with the aging 2.4 series of kernels.

The rest of the software modules operating on the experimental platform, such as the video capture device driver and the web server, were a direct result of the kernel and distribution choices.

4.3 Micro-benchmarks

In this section of our experimental evaluation, we focus on a series of micro-benchmarks to map out the basic parameters of the software and hardware homogeneous code isolation schemes used as components in our hybrid.

Micro-benchmarks are simple measures and do not reflect the performance level that any particular application would be able to obtain. Nevertheless, they are useful because they permit us to evaluate individual performance parameters in a complex and interdependent computing environment separately and to control for other variables.

The specific parameters measured and analyzed are the invocation latency and the execution overheads of plugins isolated through software and hardware techniques.

4.3.1 Latency

The invocation latency of an extension represents the delay imposed on the control transfers into and out of the extension by the features of the isolation technique employed. Invocation latencies are measured in time units and are typically more important for short running, frequently executed, or real-time extensions. In those cases these additional costs can be a significant contributor to the total extension runtime, can add up across many invocations,

or can cause the violation of deadlines, respectively.

To quantify latency overheads, we measure the runtime of a null extension, *i.e.*, one that contains no payload code and consists simply of a return statement. The simplest such null extension assembly code is shown in Listing 16.

In order to gain a more detailed understanding of the source of invocation latency, however, we also:

1. Vary the number of parameters, because dependent on the details of the isolation scheme, their number and types may impact invocation times, and

2. Inject midway instrumentation to take a time-stamp *inside* the extension where possible.

In this fashion we can differentiate and attribute costs better by accounting for parameter passing overheads, usually a responsibility of the caller, and by distinguishing between the inbound and outbound control transfer overheads. The resulting assembly code is shown in Listing 17.

As already noted, timing instrumentation adds a small yet non-zero overhead itself. Thus, in cases where the variable we are attempting to measure is comparable to or smaller than that cost, it may be impractical to perform midway instrumentation. In those cases, we opt for the usage of statistics and only measure aggregates of the variable to discount measurement costs.

Pure SFI

By design, a software fault isolated extension is 'physically' a part of the application it extends, *i.e.*, its object code is linked in its host and the two are indistinguishable from the standpoint of the operating system kernel. Their separation into disjoint faulting domains is only 'logical' and enforced by additional instrumentation injected into the extension at load time.

Because of the lack of a physical boundary, control transfers between host and extension are close to simple procedure calls within a single application. As such, they incur the absolute minimum overhead related only to the mechanics of stack switching and control

Listing 16: Minimalistic null extension lacking local stack variables and consisting solely of a single return instruction.

```
        .text
#
# int main()
#
          .
          .
          .
        cpuid                           # serialize instruction stream
        rdtsc                           # take a time-stamp
        movl    %edx,4+START_TSC        # save high bits of time-stamp
        movl    %eax,START_TSC          # save low bits of time-stamp

        push    ...                     #
          .                             # push extension parameters
          .                             #
          .                             #
        push    ...                     #
        call    extension               # invoke extension
        addl    $PARM_SIZE,%esp         # clean-up parameters' stack

        cpuid                           # serialize instruction stream
        rdtsc                           # take a time-stamp
        movl    %edx,4+END_TSC          # save high bits of time-stamp
        movl    %eax,END_TSC            # save low bits of time-stamp
          .
          .
          .

extension:                              # extension entry point
        ret                             # return instruction


        .data
START_TSC:
        .quad   0x0
END_TSC:
        .quad   0x0
```

Listing 17: Null extension and timing instrumentation including a midway time-stamp.

```
        .text
#
# int main()
#
          .
          .
          .
        cpuid                                   # serialize instruction stream
        rdtsc                                   # take a time-stamp
        movl    %edx,4+START_TSC                # save high bits of time-stamp
        movl    %eax,START_TSC                  # save low bits of time-stamp

        push    ...                             #
          .                                     # push extension parameters
          .                                     #
          .                                     #
        push    ...                             #
        call    extension                       # invoke extension
        addl    $PARM_SIZE,%esp                 # clean-up parameters' stack

        cpuid                                   # serialize instruction stream
        rdtsc                                   # take a time-stamp
        movl    %edx,4+END_TSC                  # save high bits of time-stamp
        movl    %eax,END_TSC                    # save low bits of time-stamp
          .
          .
          .


#
# void extension(...)
#
extension:                                      # extension entry point
        cpuid                                   # serialize instruction stream
        rdtsc                                   # take a time-stamp
        movl    %edx,4+MID_TSC                  # save high bits of time-stamp
        movl    %eax,MID_TSC                    # save low bits of time-stamp
        ret                                     # return instruction


        .data
START_TSC:
        .quad   0x0
MID_TSC:
        .quad   0x0
END_TSC:
        .quad   0x0
```

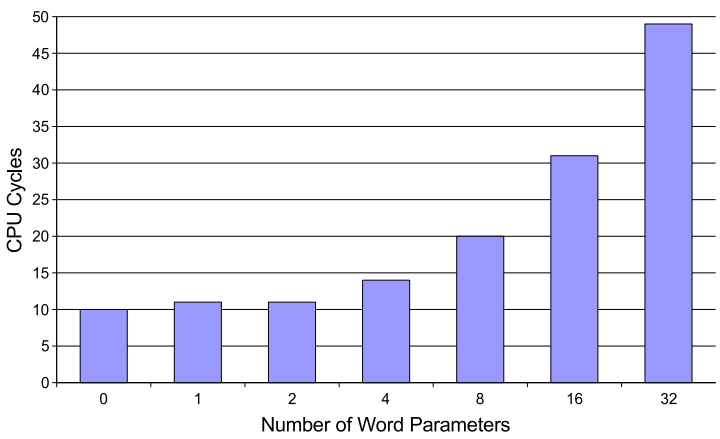

Figure 27: Null function call as a function of the number of its parameters.

transfer. Thus, they can act as a simple baseline for latency comparison across the spectrum of isolation technologies.

Listing 19 in Appendix A presents the code used to measure the null function invocation latency, and Figure 27 displays its results for our experimental Pentium III platform.

It is worth noting that the measured event is of extremely short duration, much shorter, in fact, than the duration of the timing code itself. For this reason, the code in Listing 19 is designed to sample an aggregate value for a large number of repetitions in order to amortize the expense of the measurement itself. The outer loop ensures that the actual measurement run occurs with warm data and instruction caches.

Of course, real isolation techniques are costlier and would incur a larger invocation latency penalty. We continue our evaluation by measuring our reference software fault isolation implementation, as well as other implementations for which code is available.

As previously described, our reference implementation employs a load-time code sanitizer that rewrites the untrusted input extensions' binary code so as to intercept all faults before they occur and to re-route them to separate handlers. Because the technique involves no hardware reconfiguration and no special processor setting, extensions essentially operate in the same machine environment as their host applications. This permits the actual transfer of control to occur in the same way as the simple function call from the example in Listing 19, with only the additional overhead of a stack swap and register state saving

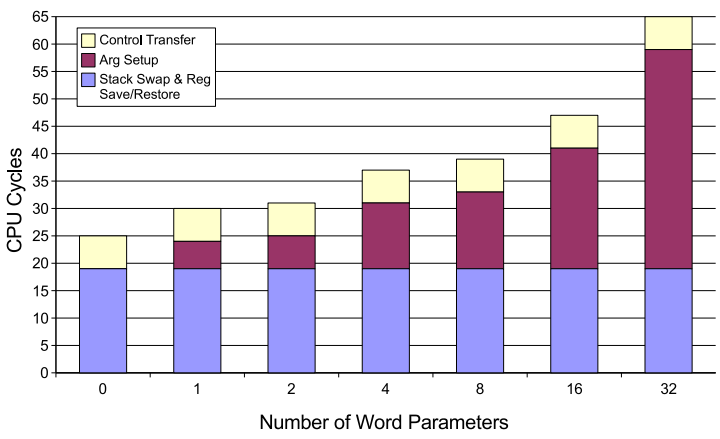

Figure 28: Invocation latency for a null software-isolated extension as a function of the number of its parameters. Includes stack swapping and state saving/restoring.

and restoring.

SFI isolated extensions live in code, data, and stack spaces that are logically separate from their hosts. Each data access, stack operation, or control transfer is checked by verification code injected in the sanitizing stage at extension load time. The IA-32 processor architecture, however, maintains pointers to the code and stack spaces, in the form of the `EIP` instruction pointer and `ESP` stack pointer registers, but no exclusive data space reference pointer exists. Thus, at each invocation the code and stack pointers need to be explicitly switched to their extension space values and then restored upon return. Additionally, as extensions are not trusted but have full access to general registers, the state of those registers must be saved and restored upon extension entry and exit, respectively.

The latter two extra steps result in a small latency increase. Listing 20 and Listing 21 in Appendix A display the codes used to measure the latency components of full software isolation, and Figure 28 shows the corresponding latency data as a function of the number of extension parameters.

Pure HFI

Our reference implementation of pure hardware fault isolation employs the memory segmentation and privilege level features of IA-32 processors. Extension memory is described

Listing 18: Modified C calling convention for extension invocation. The explicit declarations to start and list the extension's arguments permit us to place them directly in their correct places on the callee's (extension's) stack and thus avoid needless memory copying.

```
kp_call(idx, a, b, c)     =>     kp_start_args();
                                 kp_push_int(c);
                                 kp_push_int(b);
                                 kp_push_int(a);
                                 kp_call(idx);
```

by separate and mutually disjoint code and data memory regions, where the former is readable and executable and the latter is readable and writable. The extension stack, along with the global data and heap, is placed into the data segment. In addition, the descriptors for both segments place them in the processor's privilege ring 1, and thus outside of the most-privileged ring 0 where the operating system lives. Segment boundaries ensure the memory isolation property, and the lower privilege level disqualifies dangerous systems instructions, thus ensuring the code vetting property.

While affording native execution speed to extensions, this arrangement, unfortunately, complicates control transfers into and out of them by the introduction of two additional steps.

The first additional step is the copying of extension invocation parameters from the caller's stack to the separate extension stack. This is made necessary because of the difference between the privilege levels of the caller and the callee. This source of overhead can be minimized by modifying the extension calling convention and requiring the caller to explicitly place the parameters directly into a ring 0 memory mapped overlay of the extension's ring 1 stack, *e.g.*, through custom declaration as shown in Listing 18. However, as parameter copying is not the major source of overhead, this optimization provides few gains, while requiring source modifications.

The second, and more expensive, additional step involves the reconfiguration of the CPU, namely, the reloading of all architectural segment registers, in order to switch from the caller's to the callee's (extension's) address space.

Both steps contribute latency to the invocation process. Figure 29 shows the distribution of latency costs for a null extension in our hardware isolation prototype collected using

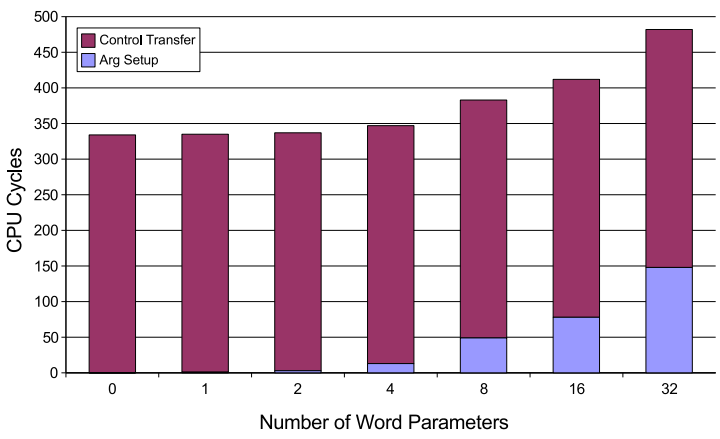

Figure 29: Invocation latency for a null hardware-isolated extension as a function of the number of its parameters. Because of the difference in caller and callee privilege level, the cost of stack swapping cannot be accounted for explicitly, as it is performed by the hardware as part of the control transfer.

the kernel module from Listing 22 in Appendix A. Remarkably, the absolute invocation latency in the purely hardware prototype is over an order of magnitude larger than its purely software counterpart. As already discussed in Chapter 3, this significant difference is a result not only of the needed CPU reconfiguration, but also of its interplay with the internal micro-architectural design of modern processors, evidenced by the disparities in the basic costs from Figure 26.

Hybrid

Improving on the purely hardware isolation, our hybrid prototype employs segmentation, but not privilege levels. Thus, both the OS kernel being extended and the extensions themselves reside in the system privilege ring 0 with no code vetting restrictions imposed by the hardware.

On the one hand, this arrangement permits us to use the faster long jumps instead of the slower interrupts and/or exceptions for inter-segment control transfer necessary for entry and exit into and out of extensions. On the other hand, however, this requires that the necessary code vetting property be provided in some alternative way.

As described previously in Chapter 3, our hybrid prototype applies a load-time sanitizer to extensions. The sanitizer operates by decoding the extension binary code, ascertaining

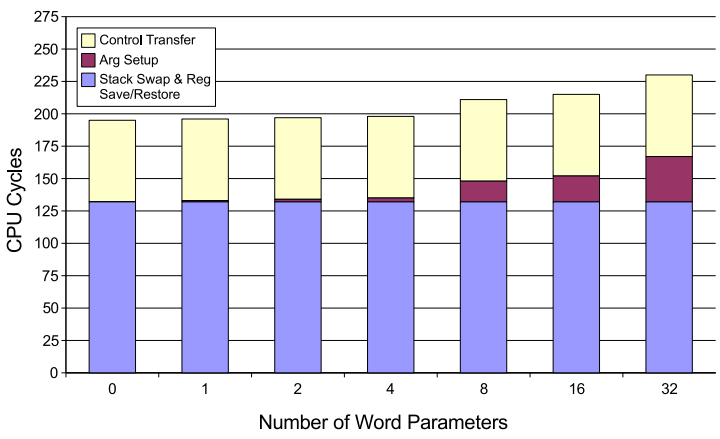

Figure 30: Invocation latency for a null hybrid-isolated extension as a function of the number of its parameters.

the absence of dangerous systems instructions, and injecting target verification code before all indirect control transfers, in order to guarantee proper machine code alignment. The resulting sanitized code satisfies the code vetting and intra-extension execution control properties, in that it is guaranteed to only branch to well-aligned instruction boundaries and to not execute any dangerous instructions. To complete the set of requirements, memory and timing isolation, as well as external execution control are respectively provided by segmentation and timer hardware.

Listing 23 in Appendix A provides the source code for the kernel modules used to quantify the invocation latency that can be achieved with our hybrid scheme. Its results are displayed in Figure 30 and show latency improvement between 48% and 58%, depending on the number of extension parameters.

Note that there is a difference in extension parameter handling between the purely hardware and hybrid prototypes. While the former copies extension parameters from the caller's stack to the callee's, the latter employs a modified calling convention in order to place the arguments directly in their correct location on the callee's stack. The implementation difference, however, does not result in significant disparities, as the cost of passing arguments, shown separately in Figure 29 and Figure 30, is dominated by the latency of control transfer and stack switching.

4.3.2 Throughput

The throughput metric reflects the execution overhead imposed on extensions isolated with a particular isolation technique. Execution overheads can be measured in two ways, either by accounting for the increase in isolated extensions' code size, or by accounting for the increase in their actual execution time. Unfortunately, neither approach is best and both have pros and cons.

The code size approach, for example, has a simple and straightforward implementation and is consistent and independent of code path and input data, in the sense that it always yields the same result when applied to the same extension. Nevertheless, the increase in an extension's instruction count is a blind measure, in that it fails to account for code structure, such as loops and conditional statements, and for the type differences in added isolation instructions. Register bound operations, for example, complete faster than memory operations. The speed of conditional control transfers varies depending on the correctness of their branch prediction and memory load/stores vary depending on whether their target hits of misses in the data cache.

The execution time approach is also simple to implement and it does account for the extension code structure and the types of isolation overhead instructions, because it measures the code's actual execution time. However, it is strongly dependent on the input data supplied to the extension and only measures the throughput reduction for the actual code path exercised by that run. Complete code path coverage for non-trivial extensions would require multiple invocations, in the general case, with specially designed input vectors to evoke them.

As a result, there is no single best way to account for the throughput reduction incurred by different isolation techniques. In the interest of completeness, then, we employ both in our real-life evaluation in Section 4.4. In this section, we use simple micro-benchmark codes with linear code path structure, so the two metrics should correlate well. Furthermore, to focus on the overheads specifically impacted by our hybrid proposal, we explore throughput reduction by studying two extensions artificially constructed to specifically exercise indirect memory references and indirect control transfers, respectively.

Because of their length, the extensions and the measurement instrumentation around them are shown in Appendix A. The indirect memory reference codes are shown in Listing 26 and Listing 27, and the indirect control transfer codes are shown in Listing 24 and Listing 25.

There are two relevant aspects to those code features – their typical frequency of occurrence in our target domain, and their isolation cost. In order to capture both into a single convenient metric, we define the term *Impact Coefficient* to be the product of the relative frequency of occurrence and the ratio of the cost of isolated over non-isolated code minus one.

$$Impact\ Coefficient = Insn\ Relative\ Frequency * (Isolated/Plain - 1) \quad (1)$$

Thus, the 'Impact Coefficient' provides us with a quantitative measure of an isolation technique's throughput overhead. Intuitively, it provides a glimpse into the "drag" an isolation technique imposes on extensions in terms relative to their instruction mix and code size. Table 4 provides an overview of the actual measurements and their computed impact coefficients.

The data support a number of interesting observations. First, the impact coefficients computed on the basis of code size and execution times correlate well, as expected from linear micro-benchmarks that do not exhibit looping or branching structure. Second, the impact coefficient of software isolation on the throughput of indirect control transfers, at approximately 8%, is relatively small. Third, the impact coefficient of software isolation on the throughput of indirect memory references, at approximately 80%, is relatively large. Finally, because hardware fault isolation executes extensions unmodified, its throughput impact coefficient would always be 0%.

Unsurprisingly, based solely on their throughput impact, it appears that if purely software isolated, typical systems code would experience a significant slowdown, whereas if purely hardware isolated, it would experience none. The key insight, however, lies in locating and demonstrating the source of the majority of SFI throughput overhead – the high frequency and high cost of isolation of indirect memory references.

That observation, in turn, supports the main premise of this dissertation, by suggesting

Table 4: Comparison of the throughput impact of different isolation techniques. Since hardware isolation techniques execute unmodified extension code, they serve as the basis for comparison. The categories for comparison are the increases in code size (bytes) and in execution time (CPU cycles) of code features subject to isolation, namely indirect memory references and indirect control transfers. The relative frequencies of appearance of such instructions are extracted from the statistical analysis of systems code appearing in Figure 21. The 'Impact Coefficient' metric is computed according to Equation 1 and is directly proportional to both usage frequency and isolation cost.

	Bytes		**Cycles**	
	Indirect Mem Ref	**Indirect Ctrl Xfer**	**Indirect Mem Ref**	**Indirect Ctrl Xfer**
Plain / HFI Isolated Code	84	102	11	43
SW Isolated Code	294	479	41	223
Insn Relative Frequency	0.31	0.02	0.31	0.02
HFI Impact Coefficient	0.00	0.00	0.00	0.00
SFI Impact Coefficient	0.78	0.07	0.85	0.08

the benefits of a hybrid solution that can handle the common case of indirect memory references in hardware and employ costly software fault isolation only for the less common case of indirect control transfers.

4.3.3 Qualitative Measures

The micro-benchmarks described above capture some of the quantitative differences between isolation techniques, but there also exist qualitative ones which can be equally or even more important. Whereas the former discriminate between performance parameters of competing techniques, the latter tend to focus on more profound distinctions that can either qualify or disqualify the usage of a particular technique.

Examples of qualitative differences can range from the ability or inability of a technique to operate on legacy extensions that are coded in a particular programming language or that are available in binary form only, through the requirement of a specific proving or trusted compiler, to the requirement for special hardware features, *e.g.*, segmentation, privilege rings, *etc.* Such qualitative differences and the associated limitations fall in the 'Adequacy'

comparison metric of our classification scheme from Chapter 3.

Because of their nature, qualitative differences are difficult to assess. Nevertheless, a hybrid technique that has been engineered in a way allowing it to revert to any of its homogeneous component techniques on the fly would always be qualitatively superior because of its inherent flexibility.

4.4 Macro-Benchmarks

Our evaluation so far has focused exclusively on micro-benchmarks focused tightly on individual performance metrics such as latency and throughput. The benefits of such simple measures are first, their ability to provide clean and uncluttered quantitative views of the performance of a complex system, and second, their ability to control unrelated variables and thus to provide experiment determinism and repeatability.

Despite their usefulness, micro-benchmarks are not sufficient for the complete evaluation of any complex system. By focusing too closely on a single facet, they fail to account for the interdependencies that exist in any such system, and as a result are not a representative of its expected performance with respect to any particular application load.

4.4.1 Motivation

In order to fill this gap in our evaluation, this section uses larger and more complex macro-benchmarks consisting of computational kernels from real-life applications. Because of their dependence on multiple, interdependent, and unpredictable variables, such as memory and cache latencies, hardware interrupts, *etc.*, we expect that these macro-benchmarks will exhibit increased variability and decreased precision.

In particular, the application kernels that we employ are the EdgeBreaker 3D triangular-mesh compression routine and a combined image processing routine consisting of an integer-only gray-scaling and edge detection algorithms demonstrated in Figure 31 and representing a transcoding engine from the CameraCast remote video sensor system [52].

These examples are chosen to satisfy a number of criteria:

Figure 31: Demonstration of the gray-scaling and edge detection image processing load. The images operated upon are in the raw PPM and PGM formats with size 640x480 pixels.

1. They represent two classes of actual application loads, as opposed to artificially constructed, and hence unrepresentative, loads,

2. They provide a convenient vehicle for gaining insight into the real-life performance characteristics that are to be expected from traditional as well as from novel hybrid isolation techniques, and finally,

3. They contribute concrete and application-specific metrics for the evaluation of the benchmark's cost, *e.g.*, the added isolation-related runtime per unit work, or performance, *e.g.*, the number of work units that can be processed by the isolated code per unit of time.

Because of their size, the listings for these benchmarks and their isolated versions are provided separately in Appendix B whereas for the remainder of this chapter we will concentrate on a discussion of their results.

4.4.2 Methodology

In order to limit the jitter induced into the benchmark measurements by external events, the codes are designed to modify the scheduling policy under which they operate to the real-time SCHED_FIFO class. Processes scheduled under it run to completion, or until they voluntarily yield the CPU or block on I/O. Additionally, SCHED_FIFO processes always take precedence over all non-real-time processes.

We execute the macro-benchmarks one at a time as the only real-time scheduled processes in the system, thus eliminating OS scheduling induced jitter. Nevertheless, the unavoidable memory caching and hardware interrupts still introduce some measurable variance into our results.

4.4.3 Aggregate Runtime

The total runtime of an actual application's kernel extension presents a practical aggregate metric of the order of performance that can be achieved on the given platform using the given isolation strategy. Figure 32 presents such aggregate runtime data for EdgeBreaker

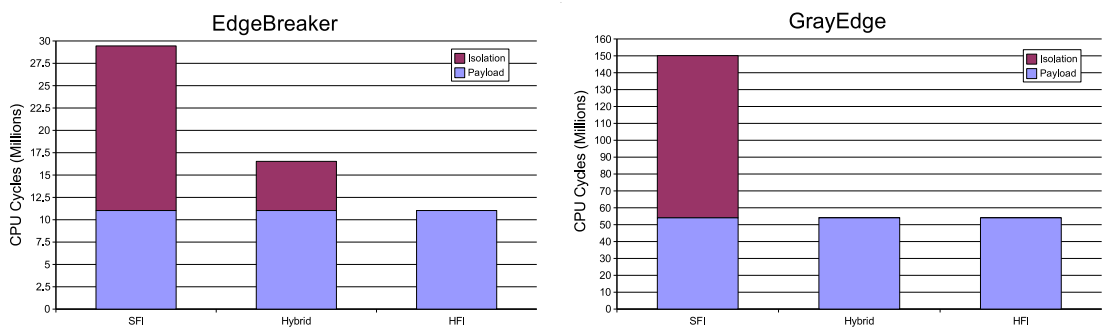

Figure 32: Breakdown of experimentally measured performance for the sample isolated real-life extensions. Note how the hybrid case of the EdgeBreaker code is somewhat costlier than predicted by the *Impact Coefficient*, whereas the hybrid case of the GrayEdge code is somewhat cheaper. Such fluctuations are caused by differences in the relative instruction composition between the respective application codes.

compression and GrayEdge image processing, our two macro-benchmarks of choice. Moreover, to aid in attributing performance costs and to emphasize their ratio, the data has been broken down into two parts, namely, the pure application load of the extension and the extra isolation overhead. The latter component includes both the latency and CPU burden dimensions of isolation cost.

While the data clearly bears out the performance part of the throughput/latency compromise for the hybrid strategy, it is interesting to note the disparity that exists in the hybrid isolation effects between the two examples. The cause for that contrast is the difference in their relative instruction composition, both with respect to each other and with respect to the "average" system code instruction composition numbers extracted from the Linux kernel and used as a basis for the impact coefficient.

The GrayEdge image code is memory-intensive and fairly monolithic, consisting of only 2 separate functions applied to the data in series. Thus, it exhibits more indirect memory references and less indirect control transfers than the average, so it is able to extract maximum benefits from the hybrid isolation strategy relative to the SFI technique.

EdgeBreaker, on the other hand, is similarly memory-intensive but rather more fragmented, consisting of 13 sub-routines, many of which are invoked a large number of times as part of its algorithm. The resulting larger share of indirect control transfers leads to a

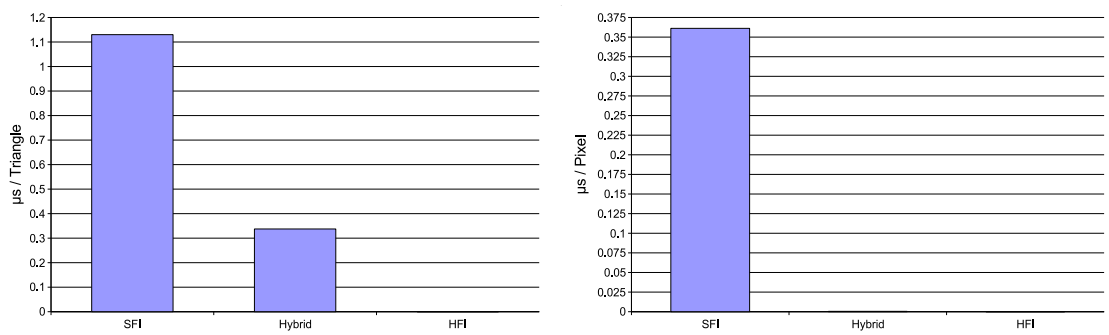

Figure 33: Experimentally measured isolation cost, presented in application-specific terms. The added cost is expressed as additional CPU μs per triangle or pixel processed, respectively.

reduction in the efficiency of the hybrid isolation strategy.

4.4.4 Application-Relative Cost & Performance

A more meaningful way of analyzing isolation induced limits is to consider them from the end application's view point. In our benchmarks, two useful application-specific cost and performance metrics are the additional runtime required to process a single triangle or pixel, respectively, and the reduction in the attainable processing rate measured in triangles or pixels per second.

Whereas the former is an application-specific relative expression of the raw aggregate measure from Figure 32, the latter is a relative metric dependent on both the absolute isolation runtime overhead and the normal non-isolated application processing performed by the extension.

It is worth noting that the cost computation for the latter in Figure 33 includes a latency cost component that is fixed and independent from the number of triangles or pixels processed. However, because it contributes a relatively small absolute term to the overall cost, it is washed out and disappears in the 'per triangle' plot for all practical purposes. Therefore, the data presented in Figure 33 can be used as a good predictor for the runtime of an application load given an input of known size.

In turn, the isolation cost can be combined with runtime data for the particular extension payload and converted into a 'performance loss' expressed in native application metrics.

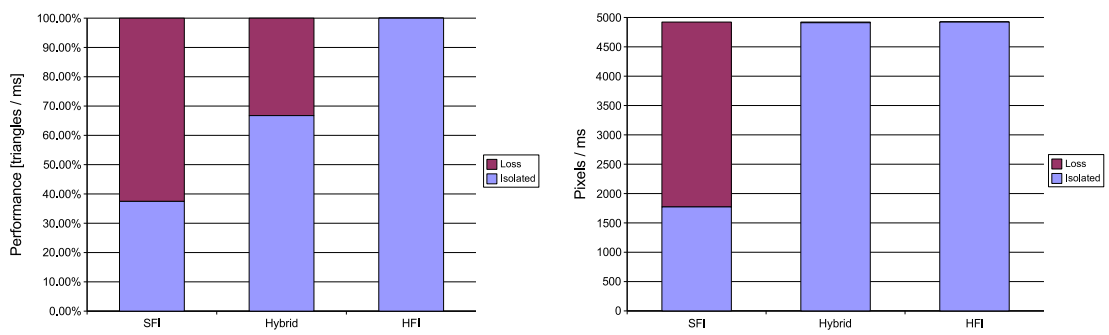

Figure 34: Experimentally measured performance loss relative to non-isolated performance. The left figure reports results for the EdgeBreaker example and the right one for GrayEdge.

Table 5: Experimentally measured performance loss relative to non-isolated performance.

Macro-Benchmark	Isolation Technique		
	SFI	**Hybrid**	**HFI**
EdgeBreaker	62.53%	33.26%	0%
GrayEdge	64.00%	0.24%	0%

For our two examples this is measured in triangles or pixels per second and demonstrates the reduction in their throughput effected by isolation. Figure 34 and Table 5 provide a relative expression of the applications' performance loss impact in graphical and tabular form, respectively.

In conclusion, we have presented an evaluation of the main classes of state-of-the-art isolation techniques. Using concrete micro- and macro-benchmarks, we have assessed their costs and demonstrated in practice that the choice of isolation technique can have significant consequence for the performance of isolated application extensions. Depending on the extensions' code composition, an unsuitable isolation technique choice can result in considerable loss of efficiency, flexibility, or both. The experimental evaluation supports the central thesis of this dissertation unequivocally, confirming initial projections of the benefits of the hybrid software/hardware approach and establishes its limits as well as some practical expectations about its average performance.

CHAPTER V

RELATED WORK

The object of this dissertation is to improve on the state-of-the-art approaches to code isolation and in this way promote extensible designs for both OS kernels and applications in general. The approach we are taking is to explore and exploit synergies among a wide variety of divergent existing techniques and combine them, or parts thereof, in order to leverage their strengths and alleviate their weaknesses. Thus, necessarily this work bears similarities with many past and present research efforts in academia as well as industry. In this chapter, we survey those related works and classify them into several categories according to their approach to ensuring the safety of isolated extension code and their relationship to the work presented in this dissertation.

5.1 Formal Methods

Formal methods represent a class of techniques that attempt to use formal safety specifications, logic reasoning, and automatic proof generation to prove invariants of extensions and safety properties about the behavior of their code.

In their essence, formal techniques aim to *prove* a piece of extension code 'safe' once and then exploit this fact every time that it is run by directly executing it within the software system that it extends, avoiding the need for any further isolation measures. There is a multitude of benefits to this approach: (1) there is no run-time overhead as the extension code is executed directly and freely within the extended application, (2) the cost of the initial proof, large though it may be, is amortized over the lifetime of the extension, and (3) formal proofs can deal not only with code isolation, but also with aspects of the code's behavior and so can additionally ensure that application invariants are preserved, for example.

Despite those desirable qualities, formal methods have not found much practical use chiefly because of the difficulty of automatic proof generation for large and/or complex codes, and to a lesser extent, the difficulty of ensuring completeness of the necessary safety

specification on which the proofs are based. Nonetheless, formal methods have been applied for simpler problems of manageable size, or to restricted problem domains. We continue this section with a description of two such formal methods: Proof-Carrying Code [77], and the Microsoft Driver Verifier [8].

5.1.1 Proof-Carrying Code

Proof-Carrying Code (PCC) [77, 74] is a mechanism through which an application can determine with certainty the safety of a binary supplied by an untrusted source. The binary is required to have been compiled with a special compiler which attaches a proof of the safety of the binary along with its object code. Safety, in this case, is defined as an attestation to the adherence of the binary code to a 'safety policy' that is provided by the application being extended. The proof takes a form that allows it to be verified easily and quickly, guarantees the behavior of its object code, and does not involve the use of encryption. At load time, the proof is validated and, barring a validation failure, it can be linked into the host application and invoked without the need for any further checks.

One of the chief advantages of the PCC approach is its total lack of run-time overhead both in terms of invocation latency and of additional checking code. Since the proof and validation process have already guaranteed the safety of a PCC extension during its compilation and loading stage, it can be executed directly and immediately when needed, without the need for latency-incurring reconfiguration of the CPU's protection hardware and without any overhead from extraneous embedded checking code.

Another major advantage is the fact that the safety policy defining the boundaries of the extension's isolated environment is a powerful approach that can also be used to define application invariants and attest the extension's adherence to them. This provides a practical means for extension *behavior* attestation that goes beyond the realm of safety and blends into correctness, *e.g.*, the ability to guarantee that for every `spin_lock()` there is a corresponding `spin_unlock()`, *etc.*

Finally, the overheads that do remain, *i.e.*, the creation and verification of the safety proof, are amortized over the lifetime and all invocations of the extension, respectively.

Once a proof is created, it can be used for as long as the extension code does not change and once verified and loaded, an extension can be invoked at will until unloaded.

The PCC approach, however, also has its disadvantages, the chief one of which is the complexity and cost of the necessary automatic proof generation. Despite advances in the technology of modern theorem provers, the computational cost of producing a PCC proof for complex and/or large extensions is still beyond practical means. Examples in the literature range from trivial functions to simple packet-filters, but fall well short of the size and complexity of useful real-life scenarios like a file-system kernel module, or a PDF browser plugin for example.

Other PCC disadvantages include its inability to provide for timing isolation by proving an upper bound for the termination of the extension. Since this is essentially the Halting Problem which has been proved NP-complete, it is unlikely that a practical formal method solution will be found. Moreover, PCC's guarantees for other isolation properties, such as Memory Isolation or Execution Control, are determined by the completeness of the safety policy.

5.1.2 SLAM and Microsoft Driver Verifier

The Microsoft Static Driver Verifier (SDV) came out of Microsoft Research's SLAM [8, 9, 7] project which aimed to apply formal reasoning techniques to detect bugs in Microsoft's popular Windows OS. The project was specifically focused on discovering bugs in device drivers as crash statistics have shown them to be the major source, accounting for roughly 85% of all newly discovered bugs [96].

The Microsoft SDV is a rule-based, compile-time, static analysis tool for device drivers that aims to test for conformance to the Windows Driver Model (WDM) API usage rules. It uses a well-defined set of 65 rules which cover 7 categories of kernel/driver interaction, ranging from the use of synchronization, to power management functions, *etc.*

The SDV performs symbolic execution to examine and evaluate the driver code without actually executing it in the context of an OS. At compile-time, it creates an abstract model of the driver that is amenable to model checking algorithms. The subject driver is first

reduced to a boolean program, where it is easier to track simple properties of variables, *e.g.*, NULL or non-NULL. The SDV then performs an exhaustive search for program states that violate its set of rules and symbolically simulates the execution path leading to those states to determine whether they are truly feasible. The SDV systematically explores all execution paths in the driver. This is difficult to do through traditional means because of the asynchronous nature of the driver environment.

While very successful at its narrowly stated goal of discovering bugs in third party drivers, the SDV approach is not general enough to address the requirements for complete code isolation. Problems like guaranteeing the safety of dynamic memory accesses or timing isolation are hard or impossible to handle with an approximating rule-based approach.

5.2 Type-Safe Languages

Another class of approaches to code isolating untrusted software components is based on the properties of type-safe programming languages. In general, these approaches restrict the implementation language and compiler to be used for both the main application and for extensions. The language is required to be type-safe, such as Modula-3, Typed Assembly Language (TAL), Java, a type-safe C dialect, *etc.* The compiler is required to be well-behaved and is trusted to perform the necessary data type checks, especially with respect to pointers. While some of those checks can be computed at compile time, others may only be computable later, during the actual runtime, so the compiler is also trusted to build additional checking instructions into the binary. Finally, the compiler typically is required to sign the binary so that its trustworthiness can be verified prior to its use.

As with other approaches, type-safe languages have their strong and weak points. On the one hand, they are able to provide code isolation efficiently by performing as much of the necessary checks as possible once during compile time. The extensions they produce merge seamlessly into applications and their use incurs no invocation latency overheads because no special treatments or hardware reconfiguration are necessary.

On the other hand, they restrict the application and extension implementation language and compiler. Not every programming language is suitable for every purpose and what is

more, in many situations such language-lock creates difficult problems with respect to legacy code or backward compatibility. Moreover, the heavy reliance on a trusted compiler means that the isolation guarantees are only as strong as the compiler is correct. Since modern compilers are quite complex, typically consisting of hundreds of thousands of lines of code, the potential for possible exploits cannot be ruled out. Finally, the additional dynamic checking code incurs some runtime overhead which is difficult to quantify, but can result in significant slowdown.

We briefly describe some recent examples of the type-safe languages used to isolate untrusted code.

5.2.1 Modula-3

Modula-3 is the type-safe language used in the SPIN [12, 92, 43, 82] extensible operating system. It supports interfaces, type safety, automatic storage management, objects, generic interfaces, threads, and exceptions, though only the first three are required for SPIN's purposes. Interfaces are used to hide implementation detail and only present a strict API to modules. The approach exploits type-safety to prevent arbitrary memory accesses by requiring that pointers only point to objects of a matching type.

These two features implement most of the execution control and memory access restriction requirements for isolation. Since they are implemented mostly at compile time, they are also mostly run-time burden-free. Some operations like dynamic array indexing, dynamic pointer casts, *etc.*, however, cannot always be statically checked at compile time. In these cases, the compiler inserts additional checking instructions. Code vetting in SPIN is realized by mandating the use of a trusted Modula-3 compiler that only generates proper code. Finally, timing isolation, unlike the rest of the isolation criteria, is unfortunately not guaranteed simply by virtue of using Modula-3. To fill this gap, SPIN implements extension preemption through a periodic timer interrupt.

5.2.2 Typed Assembly Language

Typed Assembly Language (TAL) [70, 21, 69, 40] is an extension of traditional untyped assembly languages that introduces typing annotations, memory management primitives,

and a coherent set of typing rules. These typing rules guarantee memory isolation, control isolation, and type safety of TAL programs. Timing isolation is not handled intrinsically, however, and either has to be enforced via external means or by additional code inserted by a higher-level compiler. Code vetting is directly verifiable from the machine language.

The TAL typing constructs are expressive enough to encode most source language programming features including records and structures, arrays, polymorphic functions, exceptions, abstract data types, sub-typing, and modules. TAL is also flexible enough to allow many low-level compiler optimizations.

As such, TAL itself is not so much a unique approach, but may instead be characterized as a target platform for type-safe language compilers that want to produce code that can be verified to be safe for use in extensible applications or operating system kernels. One can also look at TAL as a simple form of proof-carrying code, with the type annotations in the binary constituting the proof and the type-checker applied at load time playing the role of a proof verifier. The use of TAL frees the choice of extension implementation language to any high-level language with a TAL-target compiler and removes the requirement for trusting the compiler. In this way, TAL provides a way to relax some of the constraints of using high-level type-safe languages.

5.2.3 Type-Safe C

Recent years have seen the emergence of CCured [17, 76, 75, 37] and Cyclone [51, 34, 33, 42, 41], two new programming languages that directly address the legacy code and language lock drawbacks of type-safe languages like Modula-3, ML, Java, *etc.* They do this by attempting to address the safety problems of C, the most popular systems implementation language, instead of advocating its substitution with an alternative.

Both CCured and Cyclone extend C's type system and apply a mix of static and dynamic checks to the source code, and they are implemented as meta-compilers producing regular C language. The main focus of both is memory access control and more specifically, reining in the dangers of pointer casts and pointer arithmetic. The general approach is to add type information to pointers and then build type verification into the compiled program.

There are, however, distinct differences between the two methods. CCured is most concerned with porting legacy code with little or no change, so it infers pointer properties from the code itself and stores them in modified pointer data representations. In contrast, Cyclone is most concerned with preserving C's performance and programmer control over the low-level details of data representation and memory management, so it does not shy away from extending the language with several versions of typed pointers. Cyclone also goes beyond memory access control and integrates multi-threading, synchronization, and region-based memory management into the language.

To gain the full benefits of Cyclone, extensions still need to be ported to it from standard C, though that task is made much easier by the closeness between both languages. Its combination of low-level control, cheap porting cost, and a multitude of safety benefits has quickly popularized Cyclone and made it the language of choice for many extensible systems, such as the Open Kernel Environment (OKE) [13, 15, 14], the RBClick [83] resource management extension of the Click [68] modular router, the FLAME [6, 50, 5] programmable packet-level network monitoring architecture, *etc.*

Finally, like other programming language techniques, CCured and Cyclone are able to guarantee memory isolation, control-flow isolation, and code vetting, but cannot guarantee timing control for extensions.

5.2.4 Singularity

Probably the most novel and far-reaching language approach is a new experimental OS from Microsoft Research named Singularity [44]. Almost all of Singularity is written in a new type-safe language named Sing# which is based on C# with some important improvements. Sing# introduces the notion of message passing with semantics defined by formal contracts as an organic part of the programming language itself as opposed to an external primitive provided by a library or the OS. This helps by extending the traditional strong typing of data to the means for its transmission. All communication in Singularity is performed through Sing#'s bidirectional 'channels' which not only impose the strong typing of message content, but also requires the specification of messaging protocol through an explicit 'contract'. The

combination of these language features effectively extends the type-checking capabilities of the Sing# compiler outside of the domain of datastructures and across the wider realm of communicating processes.

Another distinguishing feature of Singularity is its novel definition of the familiar concept of a process. In traditional OSs, separate processes have always been associated with separate address spaces with the latter providing the basic foundation for inter-process isolation. The address space abstraction has invariably been implemented through some form of direct hardware support, typically an MMU, with its associated costs [4] and a large body of research on reducing them [55, 99, 16]. Singularity proposes the novel notion of 'sealed' software isolated processes, or SIPs, where the sealed part refers to a *closed process architecture* that prohibits dynamic linking and shared memory in favor of static immutable code and message passing, and the software isolation part refers to discarding the protection role of hardware (the MMU is used only for page mapping) and relying on type-isolation to enforce the walls among SIPs. The result is a safe single virtual address space system that effectively eliminates two significant sources of OS overhead – context switches and kernel traps.

Singularity's approach has the welcome characteristics of reducing common overheads and establishing formal specifications that provide a basis for rigorous reasoning about its correctness. Those characteristics, however, come at the cost of a static and immutable code model where the smallest unit of functionality is a complete SIP and where communication among SIPs is restricted only to message passing. Singularity's radical departure from traditional OS design is certainly novel and promising. It would be interesting to see how it develops and how much traction it manages to establish in the mainstream outside of the world of pure research.

5.3 Software Fault Isolation

Software Fault Isolation (SFI) [102] is a purely software technique that relies on rewriting the object code of binary extensions before they are executed. It performs static checking of memory references and control transfers that employ immediate addressing where the

targets are fixed and known *a priori*. To handle indirect addressing instructions, it inserts additional checking code into the binary that validates their actual targets at run-time. This is straightforward to do for ordinary memory references as the target address need only be checked against the boundaries of the sandbox. Control flow instructions, however, require an additional check in order to verify that their target points to the beginning of a valid instruction. This is easy on fixed instruction length architectures, as is typically the case for RISC architectures, where it translates to an additional but inexpensive alignment check. Variable instruction length architectures, usually CISC, such as the Intel IA-32 [47, 48], however, are byte-addressable and raise the possibility of a malicious or malformed branch into the middle of an instruction and thus decoding it as something different and potentially unsafe.

Two solutions for that problem have been proposed in the literature. The first one builds a table of all approved jump targets (instruction boundaries) during the initial code instrumentation step and restricts indirect control transfers only to targets listed in it. The second homogenizes the variable length CISC instruction set by padding each instruction to a longer uniform length. The length is typically chosen to be the smallest power-of-2 bytes in order to simplify the checking code.

The combination of static and dynamic checks implemented through binary rewriting allows the SFI technique to guarantee memory isolation, control flow isolation, as well as to vet untrusted code. Overhead varies depending on the particular implementation technique employed. Existing SFI implementations provide no means for timing isolation, though it is possible to augment them at some additional cost by inserting a timing check in each basic block of the untrusted code.

In terms of performance metrics, SFI has the advantages of not having any negative impact on the invocation latency of isolated code and of being universally applicable to any binary compiled with any compiler from any language and for any architecture. Its disadvantages are the requirements for an initial instrumentation pass and, much more importantly, the additional CPU burden of between 5% and 200% [91], dependent on program code, architecture, and implementation.

5.3.1 MiSFIT

MiSFIT [93, 94] is an SFI implementation for a fixed instruction length RISC MIPS [64, 65] architecture. In particular, its isolation stage is designed to operate on compiler assembly output before the final assembler converts it to object code. MiSFIT employs a look-up table to verify the validity of dynamic branch targets. The table is populated with all function entry points and labels from the source assembly. These restrictions help keep its size and look-up cost reasonable.

In principle, it is possible to apply the MiSFIT approach directly to the final object code, substituting the table look-up with a target alignment check for fixed instruction length architectures, or populating the branch-table with *all* valid instruction boundaries on variable instruction length architectures. The latter case, however, will likely result in large tables, costly branch target validation checks, and thus a significant increase in isolation overhead.

5.3.2 PittSFIeld

PittSFIeld [59, 60] stands for "Prototype IA-32 Transformation Tool for Software-based Fault Isolation Enabling Load-time Determination (of safety)" and is an SFI implementation aimed at Intel IA-32 [47, 48], a CISC variable instruction length architecture. Like MiSFIT, it operates on assembly compiler output before it is fed into the final assembler. Its approach is to modify the code by inserting no-op instructions as padding in such a way as to form 'chunks' satisfying the following properties:

1. The size and alignment of each chunk are fixed at a power of 2 bytes, equal to or larger that the length of the longest instruction encoding, *e.g.* $2^4 = 16$ bytes or larger for IA-32.

2. No instruction can straddle a chunk boundary.

3. Instructions, which could be the target of a control transfer, must always be placed at the beginning of a chunk.

The benefit of this transformation is that the isolation code for target validation of indirect branches is reduced to fast bit operations. Ensuring that control transfers only target chunk beginnings maps to checking or coercing the least-significant 4 target address bits to 0. Additionally, ensuring that control is transferred within the boundaries of a 2^n-byte sized sandbox maps to checking or coercing the $32 - n$ most-significant target address bits to the sandbox region's 'tag'.

While this keeps the overhead of the introduced checking code minimal, it also inflates the original extension's object code through the no-op instruction padding process. The latter can be thought of as inserting bubbles or stall-cycles in the super-scalar pipeline of the CPU while also effectively reducing the useful size of its instruction cache.

PittSFIeld's approach can also be applied directly to pre-compiled object code. In that case, however, as semantic information about the possible branch targets will be unknown, all instructions will have to be padded individually, transforming the real variable-length instructions into virtual new, longer but fixed-length ones and aggravating the costs of padding significantly.

5.4 Hardware Fault Isolation

Hardware-based isolation techniques exploit features of the underlying physical platform. Despite the unavoidable differences in implementation detail across different hardware platforms, they all employ similar concepts.

Hardware support for code vetting and execution control is ubiquitous in modern computer architectures and is usually based on the concept of modes of execution. There are typically two such modes, privileged and unprivileged. Potentially dangerous instructions are allowed only in the former, and mode switches between them are restricted only to a set of well-defined entry points set up by system software.

Some architectures, such as PA-RISC [39] and Intel IA-32 [46, 49] for example, provide additional intermediary modes and generalize the concept as 'privilege levels' or 'privilege rings'. The Intel IA-32 architecture is arguably the most popular such example. It provides 4 privilege levels, one of which is privileged and is typically reserved for running the OS

kernel, while the rest are unprivileged and generally run user processes. The presence of additional intermediary privilege levels can be leveraged by system software to create a hierarchy of software dependency, with every level allowed to depend only on more-privileged, and by assumption more-trustworthy, ones. Portable OS kernels like the Linux [19, 57] or Windows [80, 90] kernels, however, only use the extreme rings as that is the lowest common denominator across all modern architectures. Thus, the intermediary privilege rings have mostly been used in extensible research kernels and as a means of implementing virtualization [101, 10].

Support for timing control typically takes the form of programmable timers, such as the HPET [45], that supply periodic interruptions and form the basis for a hardware-guaranteed preemption mechanism. This mechanism is employed for timing isolation of user processes in preemptible OSs, as well as for extension code in extensible research OS kernels.

With respect to memory isolation, however, two alternative supporting hardware features exist: paging and segmentation. They offer very different approaches for achieving the same goal – checking the ultimate target of each and every memory reference issued by software.

5.4.1 Paging-Based

Paging-based memory isolation approaches rely on hardware originally built for demand-paged virtual memory. Physical memory is split into fixed sized frames, access to which can be protected as a block by means of mapping them into or unmapping them from the currently active address space. Each address space has a page table associated with it, and physical memory is mapped and unmapped through linking new frames into or unlinking existing frames from its page table. Page tables also typically provide a 'write enable/disable' bit per page, and sometimes also an 'execute enable/disable' bit per page. In addition, effective read permissions can be granted and revoked through mapping and unmapping pages. Though originally designed to support demand-paged virtual memory, those features allow the hardware to be used to implement a flexible the memory isolation scheme. Because paging hardware is practically ubiquitous, it is used to provide memory

isolation among an OS kernel and its processes in virtually all operating systems today, including Microsoft Windows, all Unix flavors, and many experimental OSs.

5.4.1.1 OS Kernels

Examples of paging-based memory isolation abound. Practically all commercial and non-research operating systems today, such as Windows [90], Linux [57], Solaris [58], *etc.*, and most experimental ones employ this technique to protect their kernels from applications as well as to enforce the walls among the latter.

They use paging hardware to implement the concept of address space, where all of the memory accessible to a process is mapped in its page table and the rest of the virtual address space is left unmapped. Each process has a separate page table, the backing store pages of which are generally disjoint from those of other processes, except for explicitly set up shared memory, and each process relies on the kernel for address space manipulations, tear-down, and switching.

As page tables tend to be large and complicated structures, CPUs employ special page table caches called Translation Lookaside Buffers (TLBs). Some architectures allow sharing of the TLB cache by 'tagging' TLB entries with an address spaces identifier, though others are untagged.

The most serious drawback to memory isolation implemented through page-based address spaces is the significant cost incurred by address space switching. In untagged TLB architectures like the Intel IA-32, the TLB needs to be flushed as part of an address space switch in order to prevent the carryover of cached page mappings into the new address space. The flush operation is costly in itself, but it also implies significant indirect costs due to the need to re-populate the TLB after each flush. Tagged TLB architectures such as MIPS, PowerPC, *etc.*, do not require such a flush, but that is compensated by increased runtime costs of address translation due to the effective TLB-size reduction because of TLB resource sharing and the resulting contention among multiple address spaces.

5.4.2 Segmentation-Based

Some modern CPUs, most notably the Intel IA-32 [46, 49] line but also others like the IBM PowerPC [105, 106], provide an alternative means of implementing memory isolation based around the concept of a memory 'segment'. Segments are described in terms of a tuple comprised of a base address, limit, type, access rights, and other properties. Multiple code and data segments can be described simultaneously, and the regions of memory to which they refer to are typically allowed to overlap.

At any given time, a set of segments are enabled by the OS kernel, and every memory reference is issued in the context of one of them. Thus, the CPU confines all software generated memory references to the currently enabled set of segments, and only privileged OS code can manipulate and enable or disable segments. Examples of the use of segmentation in addition to page-based memory isolation include the L4 [53, 56, 38] microkernel's optimized 'small spaces' [55], and Linux kernel extension schemes like Palladium [16] and Kernel Plugins [29]. Despite their well documented performance advantages [99, 29], segmentation schemes are missing from mainstream OSs largely because of the non-universal availability of segmentation hardware which complicates or precludes portability.

5.4.2.1 L4

L4 is a second generation microkernel developed at the German National Research Center for Information Technology (GMD) since 1995. It was conceived as an effort to do microkernels 'right' after the disappointing performance of the first generation of microkernels, such as L3 [54], Mach [2, 32], Chorus [35], Amoeba [97, 73], *etc.*

L4 is based on the premise that microkernels should offer only a minimal set of OS abstractions and that they can be made fast at the expense of being processor dependent and inherently non-portable. Its approach advocates an involved porting process that employs different algorithms and tailors internal datastructures to the details of the target hardware, as opposed to simply rebuilding a unified source. The L4 project's experimental evidence [53, 38, 99] not only demonstrates the viability of microkernels, but also, in a way, supports this dissertation's premise of the benefits of hybrid design. L4 can be viewed as a

hardware/hardware hybrid because of its use of both paging hardware to implement regular address spaces and segmentation hardware to implement small address spaces optimized for low-overhead context-switching.

5.4.2.2 Palladium

Palladium [16, 100] is an extensibility software architecture for user-level processes and traditional monolithic OS kernels with a prototype implemented on the popular Intel IA-32 architecture and the Linux kernel. It supports the injection of custom code into an application or into the OS kernel and thus provides a means for specializing their behavior or the services they provide.

Much like the L4 microkernel, Palladium employs both paging and segmentation in order to deal with the isolation aspects of untrusted code extensions. However, it also differs from it in that it isolates kernel extensions through a segmentation-based scheme, whereas it isolates user-level extensions though a paging-based scheme. At first blush, this functional split of Palladium's implementation resembles a hybrid approach, but it is better characterized as a union of two separate but disjoint technical solutions.

5.4.2.3 VX32

VX32 [24, 25] is a user-mode library designed to allow applications to load and execute arbitrary untrusted extensions or plugins safely by means of code isolating them through use of the Intel IA-32 segmentation hardware. The latter is used in a way similar to L4 and Palladium, except that the implementation is aimed at user-space applications and requires that extensions be processed by a compiler specially modified for the VX32 target.

5.5 Resource Control

5.5.1 RBClick

A somewhat related effort, RBClick [83, 86] proposes a hybrid approach to resource control for dynamic network router extensions. Interestingly, RBClick already employs a type-safe language (Cyclone) to guarantee the isolation and safety of the router core from untrusted extensions. However, it advocates a mixed static/dynamic accounting and verification of

router resources consumed by them. The balance for which RBClick is striving is a middle ground between static checking, which has the advantage of avoiding the problems of asynchronous termination and runtime overhead but is overly conservative, and dynamic checking, which bases its decisions on precise real-time information about the resource consumption of active code but incurs runtime overhead and could lead to an asynchronous termination of an extension.

5.6 Summary

In this chapter, we have presented many related techniques, each with a unique approach and a different set of strengths and weaknesses. To crystallize this point and to re-iterate the motivation for this dissertation's thesis, we point to a summary of the *pros* and *cons* of the major classes of techniques in an easy at-a-glance overview in Table 1 on page 50.

Some of their distinguishing features like *Latency* or *CPU Burden* are quantitative in nature and can be readily and precisely measured in unequivocal metrics. Others like *Build Cost* or *Adequacy* are more qualitative and can depend on many and diverse factors, ranging from the ability or inability of isolated code to run outside of a process context, to whether the extension can be coded in a particular programming language, to whether legacy binary-only extensions can be handled. This makes the latter harder to compare objectively, yet, nonetheless equally important.

The variability that is observed across the board and the lack of a single universally superior technique validate the rationale for our thesis and the need for a hybrid approach to code isolation.

CHAPTER VI

CONCLUSION AND FUTURE WORK

6.1 Conclusions

In this thesis we have presented a novel the design and implementation of a novel *hybrid* approach to code isolation. Our proposal is founded on the development of a classification system for safety requirements and isolation overheads and the subsequent observation and exploitation of the inverse nature of the relationship between the latency and CPU burden costs of existing software and hardware fault isolation techniques.

Our classification system provides a formal model for the exploration of the boundaries and constraints of code isolation techniques. It adopts the Universal Turing Machine as a simplified prototype of modern computing machines built with von Neumann architectures, it analyzes the model, and derives a complete set of isolation requirements in its context. The four classic properties of secure systems: confidentiality, integrity, availability, and authenticity are considered along with the corresponding attack types, resulting in the formulation of the following comprehensive set of isolation criteria:

- *Memory Isolation*: prescribes the imposition of limitations of reading and writing accesses to storage.
- *Execution Control*: stipulates the enforcement of strict constraints over the flow of control.
- *Timing Control*: dictates the establishment end enforcement of upper bounds on the runtime of untrusted code.
- *Code Vetting*: prevents the exploitation of the UTM's stateful nature by denying dangerous state transitions for isolated code.

At a technical level, this thesis surveys the state-of-the-art code isolation techniques

and compares their performance on the basis of four quantitative and qualitative metrics, namely:

- *Latency*: reflects invocation overheads.
- *CPU Burden*: reflects runtime overheads.
- *Build Cost*: reflects load-time overheads.
- *Adequacy*: reflects usability restrictions.

Implementation techniques are discussed and profiled and an inherent latency/burden trade-off is identified. Furthermore, a novel hybrid isolation technique is proposed that results in a better balance between quantitative performance metrics, as well as less restrictive and more flexible qualitative metrics.

The hybrid isolation technique employs a combination of software and hardware approaches to cover the full set of safety requirements for system extensions. It is motivated by the cost differential between the high-burden/low-latency of the former *vs.* the low-burden/high-latency of the latter and also by the relative distribution of instructions in executable machine code.

Finally, we have presented a comparative experimental evaluation of our hybrid isolation proposal that employs a number of micro-benchmarks focusing narrowly on the individual aspects of the performance trade-off, as well as two realistic macro-applications demonstrating the aggregate nature of the hybrid technique's performance.

Thus, the principal contributions of this thesis to the existing body of research are briefly summarized as:

- The development of a formal model for studying code isolation techniques, a comprehensive set of isolation requirements, and a taxonomy of metrics.
- The design and implementation of a novel hybrid isolation technique that exploits the performance trade-offs of existing approaches.

- A comparative experimental evaluation of the hybrid proposal *vs.* homogeneous alternatives using a range of micro-benchmarks as well as two realistic macro-applications.

6.2 Future Work

This thesis opens up several directions for future work, some related to the further exploration of the hybrid theme and others related to the practical usability of the current hybrid isolation code base.

An interesting direction in the former, more research-oriented, category could be the examination of possible heuristics or techniques for enabling and automating the dynamic runtime selection of isolation techniques on a case-to-case basis, perhaps guided by desired quality of service metrics.

Another direction could be to take a closer look at the changing parameters and properties of the underlying hardware platforms with an eye towards uncovering more opportunities for hybrid designs both in the quantitative performance and qualitative flexibility spaces. A concrete example of this is the disappearance of support for segmentation from the current and future generations of the Intel x86 architecture and the introduction of multi-core CPUs across the whole mobile to desktop to server space. Specifically, the built-in asymmetries of the memory hierarchy of the latter, such as cache level sharing within a CPU package for example, effectively transform them into NUMA architectures, even when computing cores are all symmetric. Moreover, the asymmetries present yet another dimension of differential performance and hence an opportunity for hybrids.

Alternatively, the latter and more practical direction for future work also presents a number of opportunities for improvement of the current prototype. The existing code is little more than a proof-of-concept and could certainly benefit a lot from a better integration between the software and the hardware fault isolation components of the hybrid. A better implementation of the binary rewriter to incorporate both flexibility in the specification of the rewriting policy and its ease of use an specification is also an alternative practical next step.

APPENDIX A

MICRO-BENCHMARK MEASUREMENT CODES

This appendix provides the listings for the latency and throughput micro-benchmark measurement codes used in the experimental evaluation in Chapter 4. They have been included here and not in the main text because of their significant length.

A.1 Latency Codes

Listing 19: Code to measure the cost of a null function call as a function of the number of its parameters.

```
#include <stdio.h>

#define REPT_SHIFT   (10)
#define REPT         (1 << REPT_SHIFT)

unsigned long long ts1;
unsigned long long ts2;

/* Test functions to be invoked */
int f0(void)
{ return 0; }

int f1(int p01)
{ return 0; }

int f2(int p01, int p02)
{ return 0; }

int f4(int p01, int p02, int p03, int p04)
{ return 0; }

int f8(int p01, int p02, int p03, int p04,
       int p05, int p06, int p07, int p08)
{ return 0; }

int f16(int p01, int p02, int p03, int p04,
        int p05, int p06, int p07, int p08,
        int p09, int p10, int p11, int p12,
        int p13, int p14, int p15, int p16)
{ return 0; }

int f32(int p01, int p02, int p03, int p04,
        int p05, int p06, int p07, int p08,
        int p09, int p10, int p11, int p12,
```

```
        int p13, int p14, int p15, int p16,
        int p17, int p18, int p19, int p20,
        int p21, int p22, int p23, int p24,
        int p25, int p26, int p27, int p28,
        int p29, int p30, int p31, int p32)
{ return 0; }

/* Macros for variable pushing */
#define PUSH0

#define PUSH1 \
  "push $0x0\n\t"

#define PUSH2 \
  PUSH1       \
  PUSH1

#define PUSH4 \
  PUSH2       \
  PUSH2

#define PUSH8 \
  PUSH4       \
  PUSH4

#define PUSH16 \
  PUSH8        \
  PUSH8

#define PUSH32 \
  PUSH16       \
  PUSH16

/* Macro to build the test driver function */
#define BUILD_BENCH(x)          \
unsigned long bench##x()        \
{                               \
    int i,j;                    \
    for(j=0; j<2; j++) {        \
      /* Initial time-stamp */  \
      __asm__ __volatile__(     \
        "cpuid\n\t"             \
        "rdtsc\n\t"             \
        : "=A" (ts1)            \
        : : "ebx", "ecx");      \
\
      /* Timing payload */      \
      for(i=0; i<=REPT; i++) {  \
        __asm__ __volatile__(   \
        PUSH##x                 \
        "call f"#x"\n\t"        \
        "addl $4*"#x",%esp\n\t" \
        );                      \
      }                         \
```

```
\
      /* Final time-stamp */     \
      __asm__ __volatile__(      \
        "cpuid\n\t"              \
        "rdtsc\n\t"              \
        : "=A" (ts2)             \
        : : "ebx", "ecx");       \
      /* Compute average */      \
      ts2 -= ts1;                \
      ts2 >>= REPT_SHIFT;        \
    }                            \
    return (unsigned long)ts2;   \
}

/* Build test driver functions */
BUILD_BENCH(0)
BUILD_BENCH(1)
BUILD_BENCH(2)
BUILD_BENCH(4)
BUILD_BENCH(8)
BUILD_BENCH(16)
BUILD_BENCH(32)

int main()
{
    printf("Null fn call as a function of number of parameters\n");

    /* Invoke test driver functions */
    printf("bench0()  = %lu\n", bench0());
    printf("bench1()  = %lu\n", bench1());
    printf("bench2()  = %lu\n", bench2());
    printf("bench4()  = %lu\n", bench4());
    printf("bench8()  = %lu\n", bench8());
    printf("bench16() = %lu\n", bench16());
    printf("bench32() = %lu\n", bench32());

    return 0;
}
```

Listing 20: Code to measure the invocation latency for a null software-isolated extension as a function of the number of its parameters.

```
#include <stdio.h>

#define REPT_SHIFT      (10)
#define REPT            (1 << REPT_SHIFT)

/* Extension stack */
unsigned char stack[1024];

/* Extension descriptor */
typedef struct {
    unsigned long saved_esp;
    unsigned long ext_esp;
```

```
} runtime_t;

runtime_t rt = { 0UL, (unsigned long)&stack[1020] };

unsigned long long ts1;
unsigned long long ts2;

/* Test functions to be invoked */
int f0(void)
{ return 0; }

int f1(int p01)
{ return 0; }

int f2(int p01, int p02)
{ return 0; }

int f4(int p01, int p02, int p03, int p04)
{ return 0; }

int f8(int p01, int p02, int p03, int p04,
       int p05, int p06, int p07, int p08)
{ return 0; }

int f16(int p01, int p02, int p03, int p04,
        int p05, int p06, int p07, int p08,
        int p09, int p10, int p11, int p12,
        int p13, int p14, int p15, int p16)
{ return 0; }

int f32(int p01, int p02, int p03, int p04,
        int p05, int p06, int p07, int p08,
        int p09, int p10, int p11, int p12,
        int p13, int p14, int p15, int p16,
        int p17, int p18, int p19, int p20,
        int p21, int p22, int p23, int p24,
        int p25, int p26, int p27, int p28,
        int p29, int p30, int p31, int p32)
{ return 0; }

/* Macros for variable pushing */
#define PUSH0

#define PUSH1 \
  "push $0x0\n\t"

#define PUSH2 \
  PUSH1       \
  PUSH1

#define PUSH4 \
  PUSH2       \
  PUSH2
```

```
#define PUSH8 \
  PUSH4       \
  PUSH4

#define PUSH16 \
  PUSH8        \
  PUSH8

#define PUSH32 \
  PUSH16       \
  PUSH16

/* Macro to build the test driver function */
#define BUILD_BENCH(x)           \
unsigned long bench##x()         \
{                                \
    int i,j;                     \
    for(j=0; j<2; j++) {         \
      /* Initial time-stamp */   \
      __asm__ __volatile__(      \
        "cpuid\n\t"              \
        "rdtsc\n\t"              \
        : "=A" (ts1)             \
        : : "ebx", "ecx");       \
\
      /* Timing payload */       \
      for(i=0; i<REPT; i++) {    \
        __asm__ __volatile__(    \
\
        "push %%ebx\n\t"         \
        "push %%ecx\n\t"         \
        "push %%edx\n\t"         \
        "push %%esi\n\t"         \
        "push %%edi\n\t"         \
        "push %%ebp\n\t"         \
        "movl %%esp,%0\n\t"      \
        "movl %1,%%esp\n\t"      \
\
        PUSH##x                  \
        "call f"#x"\n\t"         \
        "addl $4*"#x",%%esp\n\t"\
\
        "movl %0,%%esp\n\t"      \
        "pop  %%ebp\n\t"         \
        "pop  %%edi\n\t"         \
        "pop  %%esi\n\t"         \
        "pop  %%edx\n\t"         \
        "pop  %%ecx\n\t"         \
        "pop  %%ebx\n\t"         \
        : "=m" (rt.saved_esp)    \
        : "m" (rt.ext_esp));     \
      }                          \
\
      /* Final time-stamp */     \
```

```
        __asm__ __volatile__(       \
          "cpuid\n\t"               \
          "rdtsc\n\t"               \
          : "=A" (ts2)              \
          : : "ebx", "ecx");        \
        /* Compute average */       \
        ts2 -= ts1;                 \
        ts2 >>= REPT_SHIFT;         \
    }                               \
    return (unsigned long)ts2;      \
}

/* Build test driver functions */
BUILD_BENCH(0)
BUILD_BENCH(1)
BUILD_BENCH(2)
BUILD_BENCH(4)
BUILD_BENCH(8)
BUILD_BENCH(16)
BUILD_BENCH(32)

int main()
{
    printf("Null extension call as a function of number of parameters\n");

    /* Invoke test driver functions */
    printf("bench0()  = %lu\n", bench0());
    printf("bench1()  = %lu\n", bench1());
    printf("bench2()  = %lu\n", bench2());
    printf("bench4()  = %lu\n", bench4());
    printf("bench8()  = %lu\n", bench8());
    printf("bench16() = %lu\n", bench16());
    printf("bench32() = %lu\n", bench32());

    return 0;
}
```

Listing 21: Code to measure only the cost of stack swapping and state saving and restoring.

```
#include <stdio.h>

#define REPT_SHIFT      (10)
#define REPT            (1 << REPT_SHIFT)

/* Extension stack */
unsigned char stack[1024];

/* Extension descriptor */
typedef struct {
    unsigned long saved_esp;
    unsigned long ext_esp;
} runtime_t;

runtime_t rt = { 0UL, (unsigned long)&stack[1020] };
```

```
unsigned long long ts1;
unsigned long long ts2;

unsigned long bench()
{
    int i,j;
    for(j=0; j<2; j++) {
      /* Initial time-stamp */
      __asm__ __volatile__(
        "cpuid\n\t"
        "rdtsc\n\t"
        : "=A" (ts1)
        : : "ebx", "ecx");

      /* Timing payload */
      for(i=0; i<REPT; i++) {
        __asm__ __volatile__(
        "push %%ebx\n\t"
        "push %%ecx\n\t"
        "push %%edx\n\t"
        "push %%esi\n\t"
        "push %%edi\n\t"
        "push %%ebp\n\t"
        "movl %%esp,%0\n\t"
        "movl %1,%%esp\n\t"

        "movl %0,%%esp\n\t"
        "pop  %%ebp\n\t"
        "pop  %%edi\n\t"
        "pop  %%esi\n\t"
        "pop  %%edx\n\t"
        "pop  %%ecx\n\t"
        "pop  %%ebx\n\t"
        : "=m" (rt.saved_esp)
        : "m" (rt.ext_esp));
      }

      /* Final time-stamp */
      __asm__ __volatile__(
        "cpuid\n\t"
        "rdtsc\n\t"
        : "=A" (ts2)
        : : "ebx", "ecx");
      /* Compute average */
      ts2 -=  ts1;
      ts2 >>= REPT_SHIFT;
    }
    return (unsigned long)ts2;
}

int main()
{
    printf("Stack swap and state saving/restoring cost.\n");
```

```
    /* Invoke test driver functions */
    printf("bench()  = %lu\n", bench());

    return 0;
}
```

Listing 22: Kernel module code to measure the invocation latency for a null hardware-isolated extension as a function of the number of its parameters.

```
#include <linux/config.h>
#include <linux/version.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/sched.h>
#include <asm/current.h>

#include <linux/kplugins.h>

#define dbg(fmt, args...)
#define err(fmt, args...)    printk(KERN_ERR  fmt, ## args)
#define info(fmt, args...)   printk(KERN_INFO fmt, ## args)

unsigned long long t1;
unsigned long long t2;

#define REP_SHIFT    (10)
#define REPEATS      (1 << REP_SHIFT)

#ifdef  rdtsc
#undef  rdtsc
#endif
#define rdtsc(x) __asm__ __volatile__( \
                   "cpuid;rdtsc"       \
                   : "=A" (x)          \
                   : : "ebx", "ecx")

/*
 * "runtime_t" structure describing plugin space.
 */
static runtime_t * rt;

/*
 * numerical index of the registerred plugin.
 */
int fn;

asmlinkage void benchFn(runtime_t * rt)
{
  return;
}
void benchFn_end(void) {}
```

```
#define BENCHFN_LEN      ((unsigned long)benchFn_end - (unsigned long)benchFn)

void test_benchmark()
{
    int i;
    int j;
    int idx;
    unsigned long flags;

    info("test_benchmark()\n");

    /* Suspend interrupts to avoid timing interference */
    __asm__ __volatile__("pushfl; popl %0; cli\n\t" : "=g" (flags));

    benchFn(rt);
    for(j=0; j<2; j++) {
        rdtsc(t1);
        for(i=0; i<REPEATS; i++) {
            benchFn(rt);
        }
        rdtsc(t2);
        if(j) {
            t2 -=  t1;
            t2 >>= REP_SHIFT;
            info("functn avg clocks: %lu\n", (unsigned long)t2);
        }
    }

    /* Plugin 0 args */
    idx = reg_sym_bin(rt, (ptf_t)benchFn, BENCHFN_LEN, "benchFn", 1);
    if(idx < 0) {
        err("Error registerring a new dynamic (binary) function!\n");
        return;
    }
    for(j=0; j<2; j++) {
        rdtsc(t1);
        for(i=0; i<REPEATS; i++) {
            kp_call(idx, rt);
        }
        rdtsc(t2);
        if(j) {
            t2 -=  t1;
            t2 >>= REP_SHIFT;
            info("plugin,  0 args, avg clocks: %lu\n", (unsigned long)t2);
        }
    }
    if(unreg_sym(rt, "benchFn")) {
        err("Error unregisterring function!\n");
        return;
    }

    /* Plugin 1 args */
    idx = reg_sym_bin(rt, (ptf_t)benchFn, BENCHFN_LEN, "benchFn", 2);
    if(idx < 0) {
```

```
    err("Error registerring a new dynamic (binary) function!\n");
    return;
}
for(j=0; j<2; j++) {
    rdtsc(t1);
    for(i=0; i<REPEATS; i++) {
        kp_call(idx, rt, 0);
    }
    rdtsc(t2);
    if(j) {
        t2 -= t1;
        t2 >>= REP_SHIFT;
        info("plugin, 1 args, avg clocks: %lu\n", (unsigned long)t2);
    }
}
if(unreg_sym(rt, "benchFn")) {
    err("Error unregisterring function!\n");
    return;
}

/* Plugin 2 args */
idx = reg_sym_bin(rt, (ptf_t)benchFn, BENCHFN_LEN, "benchFn", 3);
if(idx < 0) {
    err("Error registerring a new dynamic (binary) function!\n");
    return;
}
for(j=0; j<2; j++) {
    rdtsc(t1);
    for(i=0; i<REPEATS; i++) {
        kp_call(idx, rt, 0, 0);
    }
    rdtsc(t2);
    if(j) {
        t2 -= t1;
        t2 >>= REP_SHIFT;
        info("plugin, 2 args, avg clocks: %lu\n", (unsigned long)t2);
    }
}
if(unreg_sym(rt, "benchFn")) {
    err("Error unregisterring function!\n");
    return;
}

/* Plugin 4 args */
idx = reg_sym_bin(rt, (ptf_t)benchFn, BENCHFN_LEN, "benchFn", 5);
if(idx < 0) {
    err("Error registerring a new dynamic (binary) function!\n");
    return;
}
for(j=0; j<2; j++) {
    rdtsc(t1);
    for(i=0; i<REPEATS; i++) {
        kp_call(idx, rt, 0, 0, 0, 0);
    }
```

```
    rdtsc(t2);
    if(j) {
        t2 -= t1;
        t2 >>= REP_SHIFT;
        info("plugin,  4 args, avg clocks: %lu\n", (unsigned long)t2);
    }
}
if(unreg_sym(rt, "benchFn")) {
    err("Error unregisterring function!\n");
    return;
}

/* Plugin 8 args */
idx = reg_sym_bin(rt, (ptf_t)benchFn, BENCHFN_LEN, "benchFn", 9);
if(idx < 0) {
    err("Error registerring a new dynamic (binary) function!\n");
    return;
}
for(j=0; j<2; j++) {
    rdtsc(t1);
    for(i=0; i<REPEATS; i++) {
        kp_call(idx, rt, 0, 0, 0, 0, 0, 0, 0, 0);
    }
    rdtsc(t2);
    if(j) {
        t2 -= t1;
        t2 >>= REP_SHIFT;
        info("plugin,  8 args, avg clocks: %lu\n", (unsigned long)t2);
    }
}
if(unreg_sym(rt, "benchFn")) {
    err("Error unregisterring function!\n");
    return;
}

/* Plugin 16 args */
idx = reg_sym_bin(rt, (ptf_t)benchFn, BENCHFN_LEN, "benchFn", 17);
if(idx < 0) {
    err("Error registerring a new dynamic (binary) function!\n");
    return;
}
for(j=0; j<2; j++) {
    rdtsc(t1);
    for(i=0; i<REPEATS; i++) {
        kp_call(idx, rt,
                0, 0, 0, 0, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0, 0, 0);
    }
    rdtsc(t2);
    if(j) {
        t2 -= t1;
        t2 >>= REP_SHIFT;
        info("plugin, 16 args, avg clocks: %lu\n", (unsigned long)t2);
    }
```

```
    }
    if(unreg_sym(rt, "benchFn")) {
        err("Error unregisterring function!\n");
        return;
    }

    /* Plugin 32 args */
    idx = reg_sym_bin(rt, (ptf_t)benchFn, BENCHFN_LEN, "benchFn", 33);
    if(idx < 0) {
        err("Error registerring a new dynamic (binary) function!\n");
        return;
    }
    for(j=0; j<2; j++) {
        rdtsc(t1);
        for(i=0; i<REPEATS; i++) {
            kp_call(idx, rt,
                    0, 0, 0, 0, 0, 0, 0, 0,
                    0, 0, 0, 0, 0, 0, 0, 0,
                    0, 0, 0, 0, 0, 0, 0, 0,
                    0, 0, 0, 0, 0, 0, 0, 0);
        }
        rdtsc(t2);
        if(j) {
            t2 -= t1;
            t2 >>= REP_SHIFT;
            info("plugin, 32 args, avg clocks: %lu\n", (unsigned long)t2);
        }
    }
    if(unreg_sym(rt, "benchFn")) {
        err("Error unregisterring function!\n");
        return;
    }

    /* Suspend interrupts to avoid timing interference */
    __asm__ __volatile__("push %0; popfl\n\t" : : "g" (flags));

    return;
}

int __init kp_demo_init(void)
{
  /* Make a runtime structure */
  rt = make_runtime("latency", 64, NULL, 4096);
  if(!rt) {
    err("Cannot allocate runtime!\n");
    return -1;
  }

  test_benchmark();

  return 0;
}

void __exit kp_demo_cleanup(void)
```

```
{
  kill_runtime(rt, 1);

  return;
}

module_init(kp_demo_init);
module_exit(kp_demo_cleanup);
```

Listing 23: Kernel module code to measure the invocation latency for a null hybrid-isolated extension as a function of the number of its parameters.

```
#include <linux/config.h>
#include <linux/version.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/sched.h>
#include <asm/current.h>
#include <asm/desc.h>

#include <linux/kplugins.h>

/*
 * "runtime_t" structure describing plugin space.
 */
static runtime_t * rt;


unsigned long flags;

unsigned long long t1;
unsigned long long t2;

#define REP_SHIFT    (10)
#define REPEATS      (1 << REP_SHIFT)

#ifdef  rdtsc
#undef  rdtsc
#endif
#define rdtsc(x) __asm__ __volatile__( \
                   "cpuid;rdtsc"       \
                   : "=A" (x)          \
                   : : "ebx", "ecx")

asmlinkage void benchFn(runtime_t * rt)
{
    return;
}
void benchFn_end(void) {}

#define BENCHFN_LEN       ((unsigned long)benchFn_end - (unsigned long)benchFn)

void test_benchmark(void)
```

```
{
    int i;
    int j;
    int idx;

    kp_info("test_benchmark()\n");

    idx = kp_reg_sym_bin(rt, (ptf_t)benchFn, BENCHFN_LEN, "benchFn");
    if(idx < 0) {
        kp_info("Error registerring a new dynamic (binary) function!\n");
        return;
    }

    /* Suspend interrupts to avoid timing interference */
    __asm__ __volatile__("pushfl; popl %0; cli\n\t" : "=g" (flags));

    benchFn(rt);
    for(j=0; j<2; j++) {
        rdtsc(t1);
        for(i=0; i<REPEATS; i++) {
            benchFn(rt);
        }
        rdtsc(t2);
        if(j) {
            t2 -= t1;
            t2 >>= REP_SHIFT;
            kp_info("functn avg clocks: %lu\n", (unsigned long)t2);
        }
    }

    /* Plugin 0 args */
    for(j=0; j<2; j++) {
        rdtsc(t1);
        for(i=0; i<REPEATS; i++) {
            kp_start_args(rt);
            kp_push_typed(rt, rt);
            kp_call(rt, idx);
        }
        rdtsc(t2);
        if(j) {
            t2 -= t1;
            t2 >>= REP_SHIFT;
            kp_info("plugin,  0 args, avg clocks: %lu\n", (unsigned long)t2);
        }
    }

    /* Plugin 1 args */
    for(j=0; j<2; j++) {
        rdtsc(t1);
        for(i=0; i<REPEATS; i++) {
            kp_start_args(rt);
            kp_push_int(rt, 0);
            kp_push_typed(rt, rt);
            kp_call(rt, idx);
```

```
    }
    rdtsc(t2);
    if(j) {
        t2 -= t1;
        t2 >>= REP_SHIFT;
        kp_info("plugin, 1 args, avg clocks: %lu\n", (unsigned long)t2);
    }
}

/* Plugin 2 args */
for(j=0; j<2; j++) {
    rdtsc(t1);
    for(i=0; i<REPEATS; i++) {
        kp_start_args(rt);
        kp_push_int(rt, 0);
        kp_push_int(rt, 0);
        kp_push_typed(rt, rt);
        kp_call(rt, idx);
    }
    rdtsc(t2);
    if(j) {
        t2 -= t1;
        t2 >>= REP_SHIFT;
        kp_info("plugin, 2 args, avg clocks: %lu\n", (unsigned long)t2);
    }
}

/* Plugin 4 args */
for(j=0; j<2; j++) {
    rdtsc(t1);
    for(i=0; i<REPEATS; i++) {
        kp_start_args(rt);
        kp_push_int(rt, 0);
        kp_push_int(rt, 0);
        kp_push_int(rt, 0);
        kp_push_int(rt, 0);
        kp_push_typed(rt, rt);
        kp_call(rt, idx);
    }
    rdtsc(t2);
    if(j) {
        t2 -= t1;
        t2 >>= REP_SHIFT;
        kp_info("plugin, 4 args, avg clocks: %lu\n", (unsigned long)t2);
    }
}

/* Plugin 8 args */
for(j=0; j<2; j++) {
    rdtsc(t1);
    for(i=0; i<REPEATS; i++) {
        kp_start_args(rt);
        kp_push_int(rt, 0);
        kp_push_int(rt, 0);
```

```
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_typed(rt, rt);
            kp_call(rt, idx);
        }
        rdtsc(t2);
        if(j) {
            t2 -= t1;
            t2 >>= REP_SHIFT;
            kp_info("plugin,  8 args, avg clocks: %lu\n", (unsigned long)t2);
        }
    }

    /* Plugin 16 args */
    for(j=0; j<2; j++) {
        rdtsc(t1);
        for(i=0; i<REPEATS; i++) {
            kp_start_args(rt);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_typed(rt, rt);
            kp_call(rt, idx);
        }
        rdtsc(t2);
        if(j) {
            t2 -= t1;
            t2 >>= REP_SHIFT;
            kp_info("plugin, 16 args, avg clocks: %lu\n", (unsigned long)t2);
        }
    }

    /* Plugin 32 args */
    for(j=0; j<2; j++) {
        rdtsc(t1);
        for(i=0; i<REPEATS; i++) {
            kp_start_args(rt);
```

```
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_int(rt, 0);
            kp_push_typed(rt, rt);
            kp_call(rt, idx);
        }
        rdtsc(t2);
        if(j) {
            t2 -= t1;
            t2 >>= REP_SHIFT;
            kp_info("plugin, 32 args, avg clocks: %lu\n", (unsigned long)t2);
        }
    }

    /* Resume interrupts */
    __asm__ __volatile__("pushl %0; popfl\n\t" : : "g" (flags));

    kp_unreg_sym(rt, "benchFn");
}

/* Module init function */
int __init kp_demo_init(void)
{
    /* Make a runtime structure */
    rt = kp_make_runtime("kp-demo", 64, 0, 0, 0);
```

```
    if(!rt) {
        kp_info("Cannot allocate runtime!\n");
        return -1;
    }

    test_benchmark();

    return 0;
}

/* Module cleanup function */
void __exit kp_demo_cleanup(void)
{
    /* Destroy the runtime structure */
    kp_kill_runtime(rt);

    return;
}

module_init(kp_demo_init);
module_exit(kp_demo_cleanup);

MODULE_AUTHOR("Ivan Ganev <ganev@cc.gatech.edu>");
MODULE_DESCRIPTION("Kernel Plugin Latency Benchmark");
#ifdef MODULE_LICENSE
MODULE_LICENSE("GPL");
#endif
```

A.2 Throughput Codes

Listing 24: Code measuring the throughput of plain indirect control transfers.

```
REPT    =       1024

.data
.align 32
ts1:
        .quad 0x0
ts2:
        .quad 0x0

.align 32
count:
        .long 0x0

.align 32
indirect:
        .long 0x0

msg1:
        .string "Inner loop bytes  = %lu\n"
msg2:
        .string "Inner loop cycles = %lu\n"
```

```
.text
        .globl main
main:

# repeat twice to warm I&D caches
        movl $2,count
outLoop:

# take start time-stamp
        cpuid
        rdtsc
        movl %eax,ts1
        movl %edx,4+ts1

# setup inner loop repetition count
        movl $REPT,%ecx

IN_BEG  =       .

# compute inner loop
inLoop:

        movl $1f,indirect
        leal indirect,%ebx
        jmp *(%ebx)
1:
        addl $9,indirect
        jmp *(%ebx)

        addl $9,indirect
        jmp *(%ebx)

        addl $9,indirect
        jmp *(%ebx)

        addl $9,indirect
        jmp *(%ebx)

        addl $9,indirect
        jmp *(%ebx)

        addl $9,indirect
        jmp *(%ebx)

        addl $9,indirect
        jmp *(%ebx)

        addl $9,indirect
        jmp *(%ebx)

        addl $9,indirect
        jmp *(%ebx)
```

```
        decl %ecx
        jnz inLoop

IN_LEN  =       . - IN_BEG

# take end time-stamp
        cpuid
        rdtsc
        movl %eax, ts2
        movl %edx,4+ts2

# loop outer loop
        decl count
        jnz outLoop

# subtract time-stamps
        movl ts2,%eax
        movl 4+ts2,%edx
        subl ts1,%eax
        sbbl 4+ts1,%edx

# divide by inner loop count
        movl $REPT, count
        divl count

# print out results
        pushl %eax
        pushl $msg2
        pushl $IN_LEN
        pushl $msg1
        call printf
        addl $0x8,%esp
        call printf
        addl $0x8,%esp

# exit
        movl $1,%eax
        movl $0,%ebx
        int  $0x80
```

Listing 25: Code measuring the throughput of software isolated indirect control transfers.

```
REPT    =       2048

.data
.align 32
ts1:
        .quad 0x0
ts2:
        .quad 0x0

.align 32
count:
        .long 0x0
```

```
.align 32
indirect:
        .long 0x0

msg1:
        .string "Inner loop bytes  = %lu\n"
msg2:
        .string "Inner loop cycles = %lu\n"

# placeholder map allowing all jumps
bitmap:
        .rept 1024
        .byte 0xff
        .endr

.text

        .globl main

main:

# repeat twice to warm I&D caches
        movl $2,count
outLoop:

# take start time-stamp
        cpuid
        rdtsc
        movl %eax,ts1
        movl %edx,4+ts1

# setup inner loop repetition count
        movl $REPT,%ecx

IN_BEG  =       .

# compute inner loop
inLoop:
        movl $1f,indirect
        leal indirect,%ebx

        pushl   %ebx
        movl    (%ebx),%ebx
        subl    $main,%ebx
        cmpl    $MAX_OFFT,%ebx
        ja      misaligned
        bt      %ebx,bitmap
        jz      misaligned
        popl    %ebx
        jmp *(%ebx)
1:
        addl $1f-.,indirect
        pushl   %ebx
```

```
        movl    (%ebx),%ebx
        subl    $main,%ebx
        cmpl    $MAX_OFFT,%ebx
        ja      misaligned
        bt      %ebx,bitmap
        jz      misaligned
        popl    %ebx
        jmp *(%ebx)
1:
        addl $1f-.,indirect
        pushl   %ebx
        movl    (%ebx),%ebx
        subl    $main,%ebx
        cmpl    $MAX_OFFT,%ebx
        ja      misaligned
        bt      %ebx,bitmap
        jz      misaligned
        popl    %ebx
        jmp *(%ebx)
1:
        addl $1f-.,indirect
        pushl   %ebx
        movl    (%ebx),%ebx
        subl    $main,%ebx
        cmpl    $MAX_OFFT,%ebx
        ja      misaligned
        bt      %ebx,bitmap
        jz      misaligned
        popl    %ebx
        jmp *(%ebx)
1:
        addl $1f-.,indirect
        pushl   %ebx
        movl    (%ebx),%ebx
        subl    $main,%ebx
        cmpl    $MAX_OFFT,%ebx
        ja      misaligned
        bt      %ebx,bitmap
        jz      misaligned
        popl    %ebx
        jmp *(%ebx)
1:
        addl $1f-.,indirect
        pushl   %ebx
        movl    (%ebx),%ebx
        subl    $main,%ebx
        cmpl    $MAX_OFFT,%ebx
        ja      misaligned
        bt      %ebx,bitmap
        jz      misaligned
        popl    %ebx
        jmp *(%ebx)
1:
        addl $1f-.,indirect
```

```
        pushl   %ebx
        movl    (%ebx),%ebx
        subl    $main,%ebx
        cmpl    $MAX_OFFT,%ebx
        ja      misaligned
        bt      %ebx,bitmap
        jz      misaligned
        popl    %ebx
        jmp *(%ebx)
1:
        addl $1f-.,indirect
        pushl   %ebx
        movl    (%ebx),%ebx
        subl    $main,%ebx
        cmpl    $MAX_OFFT,%ebx
        ja      misaligned
        bt      %ebx,bitmap
        jz      misaligned
        popl    %ebx
        jmp *(%ebx)
1:
        addl $1f-.,indirect
        pushl   %ebx
        movl    (%ebx),%ebx
        subl    $main,%ebx
        cmpl    $MAX_OFFT,%ebx
        ja      misaligned
        bt      %ebx,bitmap
        jz      misaligned
        popl    %ebx
        jmp *(%ebx)
1:
        addl $1f-.,indirect
        pushl   %ebx
        movl    (%ebx),%ebx
        subl    $main,%ebx
        cmpl    $MAX_OFFT,%ebx
        ja      misaligned
        bt      %ebx,bitmap
        jz      misaligned
        popl    %ebx
        jmp *(%ebx)
1:
        decl %ecx
        jnz inLoop

IN_LEN  =       . - IN_BEG

# end time-stamp
        cpuid
        rdtsc
        movl %eax,ts2
        movl %edx,4+ts2
```

```
# loop outer loop
        decl count
        jnz outLoop

# subtract time-stamps
        movl ts2,%eax
        movl 4+ts2,%edx
        subl ts1,%eax
        sbbl 4+ts1,%edx

# divide by inner loop count
        movl $REPT,count
        divl count

# print out results
        pushl %eax
        pushl $msg2
        pushl $IN_LEN
        pushl $msg1
        call printf
        addl $0x8,%esp
        call printf
        addl $0x8,%esp

# exit
        movl $1,%eax
        movl $0,%ebx
        int $0x80

misaligned:
        movl $1,%eax
        movl $1,%ebx
        int $0x80

MAX_OFFT =      . - main
```

Listing 26: Code measuring the throughput of plain indirect memory references.

```
REPT    =       1024

.data
.align 32
ts1:
        .quad 0x0
ts2:
        .quad 0x0

.align 32
count:
        .long 0x0

.align 32
indirect:
        .rept
```

```
        .long 0x0
        .endr

msg1:
        .string "Inner loop bytes  = %lu\n"
msg2:
        .string "Inner loop cycles = %lu\n"

.text
        .globl main
main:

# repeat twice to warm I&D caches
        movl $2,count
outLoop:

# take start time-stamp
        cpuid
        rdtsc
        movl %eax,ts1
        movl %edx,4+ts1

# setup inner loop repetition count
        movl $REPT,%ecx

IN_BEG  =       .

# compute inner loop
inLoop:

        xorl %ebx,%ebx
        movl indirect(,%ebx,4),%esi

        incl %ebx
        movl indirect(,%ebx,4),%esi

        incl %ebx
        movl indirect(,%ebx,4),%esi

        incl %ebx
        movl indirect(,%ebx,4),%esi

        incl %ebx
        movl indirect(,%ebx,4),%esi

        incl %ebx
        movl indirect(,%ebx,4),%esi

        incl %ebx
        movl indirect(,%ebx,4),%esi

        incl %ebx
        movl indirect(,%ebx,4),%esi
```

```
        incl %ebx
        movl indirect(,%ebx,4),%esi

        incl %ebx
        movl indirect(,%ebx,4),%esi

        decl %ecx
        jnz inLoop

IN_LEN  =       . - IN_BEG

# take end time-stamp
        cpuid
        rdtsc
        movl %eax, ts2
        movl %edx,4+ts2

# loop outer loop
        decl count
        jnz outLoop

# subtract time-stamps
        movl ts2,%eax
        movl 4+ts2,%edx
        subl ts1,%eax
        sbbl 4+ts1,%edx

# divide by inner loop count
        movl $REPT, count
        divl count

# print out results
        pushl %eax
        pushl $msg2
        pushl $IN_LEN
        pushl $msg1
        call printf
        addl $0x8,%esp
        call printf
        addl $0x8,%esp

# exit
        movl $1,%eax
        movl $0,%ebx
        int  $0x80
```

Listing 27: Code measuring the throughput of software isolated indirect memory references.

```
REPT            =       1024

# placeholders, used for timing purposes only
BIT_MASK        =       0x0
SEGMENT_ID      =       0x0
```

```
.data
.align 32
ts1:
        .quad 0x0
ts2:
        .quad 0x0

.align 32
count:
        .long 0x0

.align 32
indirect:
        .rept
        .long 0x0
        .endr

msg1:
        .string "Inner loop bytes  = %lu\n"
msg2:
        .string "Inner loop cycles = %lu\n"

.text
        .globl main
main:

# repeat twice to warm I&D caches
        movl $2,count
outLoop:

# take start time-stamp
        cpuid
        rdtsc
        movl %eax,ts1
        movl %edx,4+ts1

# setup inner loop repetition count
        movl $REPT,%ecx

IN_BEG  =       .

# compute inner loop
inLoop:

        xorl    %ebx,%ebx
        pushl   %eax
        leal    indirect(,%ebx,4),%eax
        andl    $BIT_MASK,%eax
        cmpl    $SEGMENT_ID,%eax
        jne     out_of_bounds
        popl    %eax
        movl    indirect(,%ebx,4),%esi
```

```
incl    %ebx
pushl   %eax
leal    indirect(,%ebx,4),%eax
andl    $BIT_MASK,%eax
cmpl    $SEGMENT_ID,%eax
jne     out_of_bounds
popl    %eax
movl    indirect(,%ebx,4),%esi

incl    %ebx
pushl   %eax
leal    indirect(,%ebx,4),%eax
andl    $BIT_MASK,%eax
cmpl    $SEGMENT_ID,%eax
jne     out_of_bounds
popl    %eax
movl    indirect(,%ebx,4),%esi

incl    %ebx
pushl   %eax
leal    indirect(,%ebx,4),%eax
andl    $BIT_MASK,%eax
cmpl    $SEGMENT_ID,%eax
jne     out_of_bounds
popl    %eax
movl    indirect(,%ebx,4),%esi

incl    %ebx
pushl   %eax
leal    indirect(,%ebx,4),%eax
andl    $BIT_MASK,%eax
cmpl    $SEGMENT_ID,%eax
jne     out_of_bounds
popl    %eax
movl    indirect(,%ebx,4),%esi

incl    %ebx
pushl   %eax
leal    indirect(,%ebx,4),%eax
andl    $BIT_MASK,%eax
cmpl    $SEGMENT_ID,%eax
jne     out_of_bounds
popl    %eax
movl    indirect(,%ebx,4),%esi

incl    %ebx
pushl   %eax
leal    indirect(,%ebx,4),%eax
andl    $BIT_MASK,%eax
cmpl    $SEGMENT_ID,%eax
jne     out_of_bounds
popl    %eax
movl    indirect(,%ebx,4),%esi
```

```
        incl    %ebx
        pushl   %eax
        leal    indirect(,%ebx,4),%eax
        andl    $BIT_MASK,%eax
        cmpl    $SEGMENT_ID,%eax
        jne     out_of_bounds
        popl    %eax
        movl    indirect(,%ebx,4),%esi

        incl    %ebx
        pushl   %eax
        leal    indirect(,%ebx,4),%eax
        andl    $BIT_MASK,%eax
        cmpl    $SEGMENT_ID,%eax
        jne     out_of_bounds
        popl    %eax
        movl    indirect(,%ebx,4),%esi

        incl    %ebx
        pushl   %eax
        leal    indirect(,%ebx,4),%eax
        andl    $BIT_MASK,%eax
        cmpl    $SEGMENT_ID,%eax
        jne     out_of_bounds
        popl    %eax
        movl    indirect(,%ebx,4),%esi

        decl %ecx
        jnz inLoop

IN_LEN  =       . - IN_BEG

# take end time-stamp
        cpuid
        rdtsc
        movl %eax, ts2
        movl %edx,4+ts2

# loop outer loop
        decl count
        jnz outLoop

# subtract time-stamps
        movl ts2,%eax
        movl 4+ts2,%edx
        subl ts1,%eax
        sbbl 4+ts1,%edx

# divide by inner loop count
        movl $REPT, count
        divl count

# print out results
        pushl %eax
```

```
        pushl $msg2
        pushl $IN_LEN
        pushl $msg1
        call  printf
        addl  $0x8,%esp
        call  printf
        addl  $0x8,%esp

# exit
        movl  $1,%eax
        movl  $0,%ebx
        int   $0x80

out_of_bounds:
        movl  $1,%eax
        movl  $1,%ebx
        int   $0x80
```

APPENDIX B

MACRO-BENCHMARK MEASUREMENT CODES

This appendix aims to provide the reader with a more complete example of how code isolation techniques are applied to an application of real-life significance. In particular, we will demonstrate the techniques on a computational kernel extracted from the Edgebreaker compression algorithm for 3D triangle mesh graphics models and an image processing engine combining gray-scaling and edge detection.

B.1 EdgeBreaker

Edgebreaker is a state-of-the-art technique, which makes it a good candidate for use and embedding in smart graphics devices and their drivers. It is more effective than generic compression techniques in reducing the amount of triangle mesh description needed in order to render a 3D object because it exploits knowledge of the internal structure of the mesh descriptions. Detailed information about the Edgebreaker algorithm can be found in [88, 89].

For purposes of this example we have extracted the compression kernel from Alla Safonova's C++ reference implementation [87] and will demonstrate isolation techniques on select functions from it. The full implementation is rather large and would inflate the size of this document needlessly and detract from the clarity of the comparison. The complete code including I/O routines and example input data is available online [87]. Next, we proceed with Listing 28 providing the C++ source code for the compression kernel from the reference implementation of the Edgebreaker algorithm.

Listing 28: C++ reference implementation of the compression kernel from the Edgebreaker triangular 3D mesh compression algorithm.

```
/************************ Types *******************************/
struct Coord3D {
  float x;
  float y;
  float z;
};
typedef Coord3D Vertex;
```

```
typedef Coord3D Vector;

#define MAX_SIZE 256

enum MeshType {MANIFOLD, TPATCH};
enum FileFormat {BINARY, ASKII};

//Variables for storing Input OVTable and geometry
extern int*     O;       //Input Opposite table
extern int*     V;       //Input Vertex indices table
extern Vertex*  G;       //Input Geometry table
extern Vertex*  G_est;   //Input Geometry table

//Compression variables
extern int T;            //triangles count
extern int N;            //vertices count
extern int *M;           //Vetex marking array
extern int *U;           //Triangles marking array

extern FileFormat eFileFormat;
void PrintErrorAndQuit(char* sErrorString);

#define fprintf(x...)
void initCompression(int c, MeshType eMeshType);
void Compress(int c);
void CheckHandle(int c);
void EncodeDelta(int c);

/********************** EB Helper Functions *****************/
int NextEdge(int edge) {
  return (3*(edge / 3) + (edge + 1) % 3);
}

int PrevEdge(int edge) {
  return NextEdge(NextEdge(edge));
}

int RightTri(int c, int* O_table) {
  //c.r = c.n.r
  return O_table[NextEdge(c)];
}

int LeftTri(int c, int* O_table) {
  //c.l = c.n.n.r
  return O_table[NextEdge(NextEdge(c))];
}

int E2T(int edge) {
  return (edge / 3);
}

/********************** EB Compression **********************/

/*
```

```
 *       Arguments:
 *       c - start compression from corner c
 *       MeshType: 2 Mesh types are currently supported:
 *                       MANIFOLD - is a manifold mesh, consistently
 *                                  oriented with no holes.
 *                       TPATCH - is a manifold mesh with boundary,
 *                                  consistently oriented.
 *       FileFormat: BINARY or ASKII (See File Formats for detais)
 *
 */
void initCompression(int c, MeshType eMeshType) {
  //init tables for marking visited vertices and triangles
  //was done in ProcessInputFile function

  //id of the last triangle compressed so far
  T = 0;

  c = PrevEdge(c);

  //estimate 1st  vertex
  EncodeDelta(NextEdge(c));

  //if we do not have a hole mark 1st  vertex as visited
  //in which case estimate function can use it for estimation
  //if we do have a hole, we do not mark 1st  vertex as visited
  //and it is not used for estimation since it is a dummy vertex
  if (eMeshType==MANIFOLD) M[V[NextEdge(c)]] = 1;


  //estimate third vertex and mark it as visited
  EncodeDelta(c);
  M[V[c]] = 1;

  //estimate second vertex and mark it as visited
  EncodeDelta(PrevEdge(c));
  M[V[PrevEdge(c)]] = 1;

  //paint the triangle
  U[E2T(c)] = 1; // mark the triangle as visited

  //traverse triangles incident on the first vertex
  //we do not want to store clers symbols for them
  int a = O[c];

  //we keep a count of number of triangles incident on the first
  //corner
  int count = 1;
  //first traverse 'C' triangles
  while (a != PrevEdge(O[PrevEdge(c)])) {
    //increment count for number of triangles incident on
    //the first corner
    count++;

    //paint the triangle, increment # of triangles
```

```
    U[E2T(a)] = 1;
    T++;

    //estimate next vertex and mark it as visited
    EncodeDelta(a);
    M[V[a]] = 1;

    //continue with the right neighbor
    a = O[NextEdge(a)];
  }

  //traverse 'R' triangle incident on first vertex
  U[E2T(a)] = 1;
  T++;
  count++;

  //write mesh type to clers file
  if (eMeshType == MANIFOLD) {
    if (eFileFormat == ASKII)
      fprintf(fclers, "%s\n", "MANIFOLD");
  } else if (eMeshType == TPATCH) {
    if (eFileFormat == ASKII)
      fprintf(fclers, "%d\n", "TPATCH");
  } else PrintErrorAndQuit("Not supported mesh type\n");

  //write number of triangles incident on first vertex to clers file
  if (eFileFormat == ASKII) fprintf(fclers, "%d\n", (int)count);

  //start connectivity compression
  Compress(O[PrevEdge(a)]);
}

void Compress(int c) {
  //start traversal for triangle tree
  do {
    //mark the triangle as visited
    U[E2T(c)] = 1;
    T++;

    //check for handles
    CheckHandle(c);

    //test whether tip vertex was visited
    if (M[V[c]] == 0) {
      //append encoding of C to clers
      fprintf(fclers, "%c\n", 'C');

      //estimate next vertex and mark it as visited
      EncodeDelta(c);
      M[V[c]] = 1;

      //continue with the right neighbor
      c = RightTri(c, O);
    } else
```

```
      //test whether right triangle was visited
      if (U[E2T(RightTri(c, O))] > 0)   {
        //test whether left triangle was visited
        if (U[E2T(LeftTri(c, O))] > 0) {
          //append code for E and pop
          fprintf(fclers, "%c\n", 'E');
          return;
        } else {
          //append code for R, move to left triangle
          fprintf(fclers, "%c\n", 'R');
          c = LeftTri(c, O);
        }
      } else
        //test whether left triangle was visited
        if (U[E2T(LeftTri(c, O))] > 0) {
          //append code for L, move to right triangle
          fprintf(fclers, "%c\n", 'L');
          c = RightTri(c, O);
        } else {
          //store corner number in decompression, to support handles
          U[E2T(c)] = T*3+2;

          //append code for S
          fprintf(fclers, "%c\n", 'S');

          //recursive call to visit right branch first
          Compress(RightTri(c, O));

          //move to left triangle
          c = LeftTri(c, O);

          //if the triangle to the left was visited, then  return
          if (U[E2T(c)]>0) return;
        }
  } while(true);
}

void CheckHandle(int c) {
  //check for handles from the right
  if (U[E2T(O[NextEdge(c)])] >1) {
    //write opposite corners for handle triangles into file
    fprintf(fhandles, "%d %d\n", U[E2T(O[NextEdge(c)])], T*3+1);
  }

  //check for handles from the left
  if (U[E2T(O[PrevEdge(c)])] >1) {
    //write opposite corners for handle triangles into file
    fprintf(fhandles, "%d %d\n", U[E2T(O[PrevEdge(c)])], T*3+2);
  }
}

/************ Vector Operations for Estimate functions *********/
//Returns v1 - v2
Vector VMinus(Vertex v1, Vertex v2) {
```

```
  Vector tempVector;
  tempVector.x = v1.x - v2.x;
  tempVector.y = v1.y - v2.y;
  tempVector.z = v1.z - v2.z;
  return tempVector;
}

//Returns v1 + v2
Vector VPlus(Vertex v1, Vector v2) {
  Vector tempVector;
  tempVector.x = v2.x + v1.x;
  tempVector.y = v2.y + v1.y;
  tempVector.z = v2.z + v1.z;
  return tempVector;
}

//Returns v1*k
Vector VMult(Vertex v1, float k) {
  Vector tempVector;
  tempVector.x = v1.x*k;
  tempVector.y = v1.y*k;
  tempVector.z = v1.z*k;
  return tempVector;
}

/********************** Estimate functions ******************/
/*
 * This function does not do any prediction, it just writes
 * vertices into array
 */
void EncodeNoPrediction(int c) {
  //Store vertex coordinates into file
  fprintf(fvertices, "%f %f %f\n", G[V[c]].x, G[V[c]].y, G[V[c]].z);
}

void EncodeWithPrediction(int c) {
  Vector vPred, delta;
  Vertex zeroV = {0.0, 0.0, 0.0};

  if (M[V[O[c]]] > 0 && M[V[PrevEdge(c)]] > 0) {
    vPred = VPlus(G_est[V[NextEdge(c)]], G_est[V[PrevEdge(c)]]);
    vPred = VMinus(vPred, G_est[V[O[c]]]);
    delta = VMinus(G[V[c]], vPred);
    //return vPred;
  } else if (M[V[O[c]]] > 0) {
    vPred = VMult(G_est[V[NextEdge(c)]], 2);
    vPred = VMinus(vPred, G_est[V[O[c]]]);
    delta = VMinus(G[V[c]], vPred);
    //return vPred;
  } else if (M[V[NextEdge(c)]] > 0 && M[V[PrevEdge(c)]] > 0) {
    vPred = VPlus(G_est[V[NextEdge(c)]], G_est[V[PrevEdge(c)]]);
    vPred = VMult(vPred, 0.5f);
    delta = VMinus(G[V[c]], vPred);
    //return vPred;
```

```
  } else if (M[V[NextEdge(c)]] > 0) {
    vPred = G_est[V[NextEdge(c)]];
    delta = VMinus(G[V[c]], vPred);
    //return vPred;
  } else if (M[V[PrevEdge(c)]] > 0) {
    vPred = G_est[V[PrevEdge(c)]];
    delta = VMinus(G[V[c]], vPred);
    //return vPred;
  } else {
    vPred = zeroV;
    delta = VMinus(G[V[c]], vPred);
  }

  G_est[V[c]] = VPlus(delta, vPred);

  fprintf(fvertices, "%f %f %f\n", delta.x, delta.y, delta.z);
}

void EncodeDelta(int c) {
  EncodeNoPrediction(c);
  //EncodeWithPrediction(c);
}
```

Even though the C++ source code seems short, the compression kernel results in approximately 1300 lines of assembly code, so it is impractical to include in this document. In order to further narrow it down to a size small enough for inclusion we profiled it to identify a suitable high-impact subset of functions subject to the following requirements:

1. The subset should be computationally representative of the workload in terms of the number of total instructions executed,

2. The subset should capture interesting behavior, such as cache misses,

3. The subset should translate into assembly code short enough to include in this document.

We ran the Edgebreaker compression kernel through the Cachegrind [78, 79] cache profiler while compressing a sample 5804 triangle 3D mesh model downloaded along with the reference code and shown in Figure 35. The resulting statistics are displayed in Table 6, where:

- **Ir** – instructions executed,

Figure 35: An example 3D triangular mesh model consisting of a 5804 triangles.

- **Dr** – data reads,
- **D1mr** – L1 data cache read misses,
- **D2mr** – L2 data cache read misses,
- **Dw** – data writes,
- **D1mw** – L1 data cache write misses,
- **D2mw** – L2 data cache write misses.

The profile data was collected on an 866 MHz Intel Pentium III with 16 KB 4-way set-associative L1 I&D caches and a 256 KB 8-way set-associative unified L2 cache. Cacheline size is 32 bytes.

Despite the fact that the `NextEdge()` routine contains the largest portion of instructions executed, the function itself is simple – consisting of a single line of code – and does not result in interesting cache behavior. Similarly, most of the other functions are either unremarkable or concerned with I/O, with the exception of `Compress()` and `CheckHandle()`. We choose the latter as examples to focus on for the rest of this chapter because they combine all of the required features – significant contributions to the total compute load (Ir), interesting cache behavior (D1mr, D2mr, D1mw, D2mw), and relatively small yet non-trivial implementations.

We continue with a contrasting view between the non-sandboxed and the PittSFIeld-sandboxed assembly codes for the two chosen routines displayed in Listing 29, and finally, a

Table 6: Cache profile generated with Cachegrind for the Edgebreaker compression kernel operating on the triangular 3D mesh from Figure 35.

Ir	Dr	D1mr	D2mr	Dw	D1mw	D2mw	Function
1,036,980	207,396	0	0	138,264	0	0	NextEdge()
330,964	55,160	1	0	113,227	1,089	1,087	ProcessInputFile()
256,993	70,089	0	0	23,363	0	0	E2T()
217,916	68,119	5,087	257	114,521	22	1	Compress()
179,800	69,600	6,834	316	69,600	1	0	CheckHandle()
66,600	27,750	12	0	27,750	0	0	LeftTri()
60,984	29,040	2,092	44	20,328	0	0	EncodeNoPrediction()
60,510	30,255	21	0	18,153	0	0	RightTri()
52,254	11,612	0	0	23,224	0	0	PrevEdge()
11,616	2,904	0	0	2,904	0	0	EncodeDelta()
136	50	6	2	62	7	0	initCompression()
78	45	4	4	26	4	2	ProcessArguments()
45	10	0	0	17	0	0	ClearMemoryAndFiles()
27	8	3	3	13	2	1	main()
23	2	0	0	13	2	2	OpenOutputFiles()

similar contrasting view between the non-sandboxed and hybrid-sandboxed assembly codes in Listing 30.

It is worthwhile to note that while the code inflation cost evident in the PittSFIeld comparison is relatively modest, it is reliant on the use, behavior, and correctness, of a particular compiler, *gcc* in this case, as well as on the presence of the original source code. Relaxing these constraints would increase the applicability of the technique, possibly extending its use to binary legacy code, but it would also aggravate its code inflation effect. When semantic information of the legal control transfer points is lost, any instruction boundary could be one and thus every instruction must be padded into its own chunk, resulting in much more significant inefficiencies.

Listing 29: Contrast between the non-sandboxed and the PittSFIeld-sandboxed assemblies for the `CheckHandle()` and `Compress()` routines from the Edgebreaker compression reference implementation. Note that actual code inflation is not necessarily proportional to assembly length, as the inserted alignment directives can translate to long or multiple instructions, *e.g.*, up to CHUNK_SIZE-1 bytes.

```
;; unsandboxed
;; CheckHandle()

    .align 2
    .p2align 4,,15
.globl _Z11CheckHandlei
```

```
;; PittSFIeld sandboxed
;; CheckHandle()

    .align 2
    .p2align 4,,15
.globl _Z11CheckHandlei
```

```
_Z11CheckHandlei:
.LFB114:
.LVL13:
    pushl   %ebp
.LCFI40:
.LBB93:
.LBB94:
    movl    $1431655766, %eax
.LBE94:
.LBE93:
    movl    %esp, %ebp
.LCFI41:
    pushl   %edi
.LCFI42:
    pushl   %esi
.LCFI43:
    subl    $32, %esp
.LCFI44:
    movl    8(%ebp), %esi
.LBB95:
.LBB96:
    imull   %esi
    movl    %esi, %ecx
    sarl    $31, %ecx
    addl    $1, %esi
    movl    $1431655766, %eax
    movl    %esi, -20(%ebp)
    subl    %ecx, %edx
    movl    -20(%ebp), %ecx
    leal    (%edx,%edx,2), %edi
    imull   %esi
.LBE96:
.LBE95:
    movl    0, %eax
.LBB97:
.LBB98:
    sarl    $31, %ecx
    movl    %edx, %esi
    subl    %ecx, %esi
    leal    (%esi,%esi,2), %ecx
    subl    %ecx, -20(%ebp)
    movl    -20(%ebp), %ecx
.LBE98:
.LBE97:
    movl    %eax, -16(%ebp)
.LBB99:
.LBB100:
    leal    (%edi,%ecx), %esi
.LBE100:
.LBE99:
    movl    (%eax,%esi,4), %edi
    movl    $1431655766, %eax
    imull   %edi
    movl    U, %eax
```

```
    .p2align 4
_Z11CheckHandlei:
.LFB114:
.LVL13:
    pushl   %ebp
.LCFI40:
.LBB93:
.LBB94:
    movl    $1431655766, %eax
.LBE94:
.LBE93:
    movl    %esp, %ebp
.LCFI41:
    pushl   %edi
.LCFI42:
    pushl   %esi
.LCFI43:
    subl    $32, %esp
.LCFI44:
    .p2align 4
    movl    8(%ebp), %esi
.LBB95:
.LBB96:
    imull   %esi
    movl    %esi, %ecx
    sarl    $31, %ecx
    .p2align 4
    andl    $0x20ffffff, %esp
    addl    $1, %esi
    movl    $1431655766, %eax
    .p2align 4
    movl    %esi, -20(%ebp)
    subl    %ecx, %edx
    movl    -20(%ebp), %ecx
    leal    (%edx,%edx,2), %edi
    imull   %esi
.LBE96:
.LBE95:
    .p2align 4
    movl    0, %eax
.LBB97:
.LBB98:
    sarl    $31, %ecx
    movl    %edx, %esi
    subl    %ecx, %esi
    leal    (%esi,%esi,2), %ecx
    .p2align 4
    subl    %ecx, -20(%ebp)
    movl    -20(%ebp), %ecx
.LBE98:
.LBE97:
    movl    %eax, -16(%ebp)
.LBB99:
.LBB100:
```

```
    sarl    $31, %edi
    movl    %edx, %ecx
    subl    %edi, %ecx
    movl    (%eax,%ecx,4), %edx
    movl    %eax, -12(%ebp)
    cmpl    $1, %edx
    jle     .L43
.LVL14:
    movl    T, %eax
    movl    %edx, 8(%esp)
    movl    $.LC6, 4(%esp)
    leal    1(%eax,%eax,2), %eax
    movl    %eax, 12(%esp)
    movl    fhandles, %eax
    movl    %eax, (%esp)
    call    fprintf
    movl    O, %edx
    movl    U, %ecx
    movl    %edx, -16(%ebp)
    movl    %ecx, -12(%ebp)
.L43:
    movl    $1431655766, %eax
    imull   %esi
    movl    %esi, %eax
    sarl    $31, %eax
    leal    1(%esi), %ecx
    movl    %edx, %edi
    subl    %eax, %edi
    movl    $1431655766, %eax
    imull   %ecx
    movl    %ecx, %eax
    sarl    $31, %eax
    movl    %edx, %esi
    subl    %eax, %esi
    leal    (%esi,%esi,2), %eax
    subl    %eax, %ecx
    leal    (%edi,%edi,2), %eax
    addl    %ecx, %eax
    movl    -16(%ebp), %ecx
    movl    (%ecx,%eax,4), %esi
    movl    $1431655766, %eax
    imull   %esi
    movl    -12(%ebp), %eax
    sarl    $31, %esi
    movl    %edx, %ecx
    subl    %esi, %ecx
    movl    (%eax,%ecx,4), %edx
    cmpl    $1, %edx
    jle     .L47
    movl    T, %eax
    movl    %edx, 8(%esp)
    movl    $.LC6, 4(%esp)
    leal    2(%eax,%eax,2), %eax
    movl    %eax, 12(%esp)
    leal    (%edi,%ecx), %esi
.LBE100:
.LBE99:
    .p2align 4
    movl    (%eax,%esi,4), %edi
    movl    $1431655766, %eax
    imull   %edi
    .p2align 4
    movl    U, %eax
    sarl    $31, %edi
    movl    %edx, %ecx
    subl    %edi, %ecx
    movl    (%eax,%ecx,4), %edx
    .p2align 4
    movl    %eax, -12(%ebp)
    cmpl    $1, %edx
    jle     .L43
.LVL14:
    .p2align 4
    movl    T, %eax
    movl    %edx, 8(%esp)
    .p2align 4
    movl    $.LC6, 4(%esp)
    leal    1(%eax,%eax,2), %eax
    movl    %eax, 12(%esp)
    .p2align 4
    movl    fhandles, %eax
    movl    %eax, (%esp)
    .p2align 4
    .byte 0x8d, 0xb4, 0x26, 0, 0, 0, 0
    .byte 0x8d, 0x74, 0x26, 0
    call    fprintf
    movl    O, %edx
    movl    U, %ecx
    movl    %edx, -16(%ebp)
    .p2align 4
    movl    %ecx, -12(%ebp)
    .p2align 4
.L43:
    movl    $1431655766, %eax
    imull   %esi
    movl    %esi, %eax
    sarl    $31, %eax
    .p2align 4
    leal    1(%esi), %ecx
    movl    %edx, %edi
    subl    %eax, %edi
    movl    $1431655766, %eax
    imull   %ecx
    .p2align 4
    movl    %ecx, %eax
    sarl    $31, %eax
    movl    %edx, %esi
    subl    %eax, %esi
```

```
    movl    fhandles, %eax
    movl    %eax, (%esp)
    call    fprintf
.L47:
    addl    $32, %esp
    popl    %esi
    popl    %edi
    popl    %ebp
    ret
.LFE114:
```

```
    leal    (%esi,%esi,2), %eax
    subl    %eax, %ecx
    .p2align 4
    leal    (%edi,%edi,2), %eax
    addl    %ecx, %eax
    movl    -16(%ebp), %ecx
    movl    (%ecx,%eax,4), %esi
    .p2align 4
    movl    $1431655766, %eax
    imull   %esi
    movl    -12(%ebp), %eax
    sarl    $31, %esi
    .p2align 4
    movl    %edx, %ecx
    subl    %esi, %ecx
    movl    (%eax,%ecx,4), %edx
    cmpl    $1, %edx
    jle     .L47
    .p2align 4
    movl    T, %eax
    movl    %edx, 8(%esp)
    .p2align 4
    movl    $.LC6, 4(%esp)
    leal    2(%eax,%eax,2), %eax
    movl    %eax, 12(%esp)
    .p2align 4
    movl    fhandles, %eax
    movl    %eax, (%esp)
    .p2align 4
    .byte 0x8d, 0xb4, 0x26, 0, 0, 0, 0
    .byte 0x8d, 0x74, 0x26, 0
    call    fprintf
    .p2align 4
.L47:
    addl    $32, %esp
    popl    %esi
    popl    %edi
    popl    %ebp
    andl    $0x20ffffff, %ebp
    .p2align 4
    andl    $0x10fffff0, (%esp)
    ret
.LFE114:
```

```
;; unsandboxed
;; Compress()

    .align 2
    .p2align 4,,15
.globl _Z8Compressi
_Z8Compressi:
.LFB8:
.LVL22:
```

```
;; PittSFIeld sandboxed
;; Compress()

    .align 2
    .p2align 4,,15
.globl _Z8Compressi
    .p2align 4
_Z8Compressi:
.LFB8:
```

```
    pushl   %ebp
.LCFI39:
    movl    %esp, %ebp
.LCFI40:
    pushl   %edi
.LCFI41:
    pushl   %esi
.LCFI42:
    subl    $44, %esp
.LCFI43:
    movl    U, %eax
    movl    V, %ecx
    movl    O, %edx
    movl    8(%ebp), %edi
    movl    %eax, -36(%ebp)
    movl    %ecx, -40(%ebp)
    movl    %eax, %ecx
    movl    %edx, -32(%ebp)
    jmp     .L43
.LVL23:
    .p2align 4,,7
.L54:
    movl    $1, (%eax)
.LBB379:
.LBB380:
    movl    -20(%ebp), %eax
    movl    (%eax), %ecx
.L45:
.LBE380:
.LBE379:
    movl    %ecx, %edi
    movl    -36(%ebp), %ecx
.LVL24:
.L43:
.LBB381:
.LBB382:
    movl    $1431655766, %eax
    imull   %edi
    movl    %edi, %eax
    sarl    $31, %eax
    subl    %eax, %edx
.LBE382:
.LBE381:
    leal    (%ecx,%edx,4), %ecx
    movl    $1, (%ecx)
    movl    T, %eax
.LBB383:
.LBB384:
.LBB385:
.LBB386:
    leal    (%edx,%edx,2), %esi
    leal    1(%edi), %edx
    movl    %edx, -48(%ebp)
.LBE386:
```

```
.LVL22:
    pushl   %ebp
.LCFI39:
    movl    %esp, %ebp
.LCFI40:
    pushl   %edi
.LCFI41:
    pushl   %esi
.LCFI42:
    subl    $44, %esp
.LCFI43:
    movl    U, %eax
    .p2align 4
    movl    V, %ecx
    movl    O, %edx
    movl    8(%ebp), %edi
    .p2align 4
    movl    %eax, -36(%ebp)
    movl    %ecx, -40(%ebp)
    movl    %eax, %ecx
    movl    %edx, -32(%ebp)
    .p2align 4
    andl    $0x20ffffff, %esp
    jmp     .L43
.LVL23:
    .p2align 4,,7
    .p2align 4
.L54:
    leal    (%eax), %ebx
    .p2align 4
    andl    $0x20ffffff, %ebx
    movl    $1, (%ebx)
.LBB379:
.LBB380:
    movl    -20(%ebp), %eax
    .p2align 4
    movl    (%eax), %ecx
    .p2align 4
.L45:
.LBE380:
.LBE379:
    movl    %ecx, %edi
    movl    -36(%ebp), %ecx
.LVL24:
    .p2align 4
.L43:
.LBB381:
.LBB382:
    movl    $1431655766, %eax
    imull   %edi
    movl    %edi, %eax
    sarl    $31, %eax
    subl    %eax, %edx
.LBE382:
```

```
.LBE385:
.LBE384:
.LBE383:
    movl    %ecx, -28(%ebp)
.LBB387:
.LBB388:
.LBB389:
.LBB390:
    movl    -48(%ebp), %ecx
.LBE390:
.LBE389:
.LBE388:
.LBE387:
    addl    $1, %eax
    movl    %eax, T
    movl    %eax, -24(%ebp)
.LBB391:
.LBB392:
.LBB393:
.LBB394:
    movl    $1431655766, %eax
    imull   %edx
.LBE394:
.LBE393:
    movl    -32(%ebp), %eax
.LBB395:
.LBB396:
    sarl    $31, %ecx
    subl    %ecx, %edx
    leal    (%edx,%edx,2), %edx
    subl    %edx, -48(%ebp)
    movl    -48(%ebp), %edx
    leal    (%esi,%edx), %ecx
.LBE396:
.LBE395:
    leal    (%eax,%ecx,4), %eax
    movl    (%eax), %edx
    movl    %eax, -20(%ebp)
    movl    $1431655766, %eax
    movl    %edx, -12(%ebp)
    imull   %edx
    movl    -12(%ebp), %eax
    sarl    $31, %eax
    subl    %eax, %edx
    movl    -36(%ebp), %eax
    movl    (%eax,%edx,4), %edx
    movl    $1431655766, %eax
    movl    %edx, -16(%ebp)
    imull   %ecx
    movl    %ecx, %eax
    sarl    $31, %eax
    addl    $1, %ecx
    subl    %eax, %edx
    movl    $1431655766, %eax
```

```
.LBE381:
    .p2align 4
    leal    (%ecx,%edx,4), %ecx
    leal    (%ecx), %ebx
    .p2align 4
    andl    $0x20ffffff, %ebx
    movl    $1, (%ebx)
    .p2align 4
    movl    T, %eax
.LBB383:
.LBB384:
.LBB385:
.LBB386:
    leal    (%edx,%edx,2), %esi
    leal    1(%edi), %edx
    movl    %edx, -48(%ebp)
.LBE386:
.LBE385:
.LBE384:
.LBE383:
    .p2align 4
    movl    %ecx, -28(%ebp)
.LBB387:
.LBB388:
.LBB389:
.LBB390:
    movl    -48(%ebp), %ecx
.LBE390:
.LBE389:
.LBE388:
.LBE387:
    addl    $1, %eax
    movl    %eax, T
   .p2align 4
   movl    %eax, -24(%ebp)
.LBB391:
.LBB392:
.LBB393:
.LBB394:
    movl    $1431655766, %eax
    imull   %edx
.LBE394:
.LBE393:
    movl    -32(%ebp), %eax
.LBB395:
.LBB396:
    .p2align 4
    sarl    $31, %ecx
    subl    %ecx, %edx
    leal    (%edx,%edx,2), %edx
    subl    %edx, -48(%ebp)
    movl    -48(%ebp), %edx
    .p2align 4
    leal    (%esi,%edx), %ecx
```

```
    leal    (%edx,%edx,2), %esi
    imull   %ecx
    movl    %ecx, %eax
    sarl    $31, %eax
    subl    %eax, %edx
    movl    $1431655766, %eax
    leal    (%edx,%edx,2), %edx
    subl    %edx, %ecx
    movl    -32(%ebp), %edx
    addl    %ecx, %esi
    movl    (%edx,%esi,4), %ecx
.LVL25:
    imull   %ecx
    movl    %ecx, %eax
    sarl    $31, %eax
    subl    %eax, %edx
    movl    -36(%ebp), %eax
    movl    (%eax,%edx,4), %edx
    movl    %edx, -44(%ebp)
.LBE392:
.LBE391:
    movl    -40(%ebp), %edx
    movl    (%edx,%edi,4), %eax
    sall    $2, %eax
    addl    M, %eax
    movl    (%eax), %edx
    testl   %edx, %edx
    je  .L54
    movl    -16(%ebp), %eax
    testl   %eax, %eax
    jle .L48
    movl    -44(%ebp), %eax
    testl   %eax, %eax
    jle .L45
.L52:
    addl    $44, %esp
    popl    %esi
    popl    %edi
.LVL26:
    popl    %ebp
    ret
.LVL27:
    .p2align 4,,7
.L48:
    movl    -44(%ebp), %eax
    movl    -12(%ebp), %ecx
    testl   %eax, %eax
    jg  .L45
    movl    -24(%ebp), %edx
    movl    -28(%ebp), %ecx
.LVL28:
    leal    2(%edx,%edx,2), %eax
    movl    -20(%ebp), %edx
    movl    %eax, (%ecx)
.LBE396:
.LBE395:
    leal    (%eax,%ecx,4), %eax
    movl    (%eax), %edx
    movl    %eax, -20(%ebp)
    .p2align 4
    movl    $1431655766, %eax
    movl    %edx, -12(%ebp)
    imull   %edx
    movl    -12(%ebp), %eax
    .p2align 4
    sarl    $31, %eax
    subl    %eax, %edx
    movl    -36(%ebp), %eax
    movl    (%eax,%edx,4), %edx
    .p2align 4
    movl    $1431655766, %eax
    movl    %edx, -16(%ebp)
    imull   %ecx
    movl    %ecx, %eax
    .p2align 4
    sarl    $31, %eax
    addl    $1, %ecx
    subl    %eax, %edx
    movl    $1431655766, %eax
    .p2align 4
    leal    (%edx,%edx,2), %esi
    imull   %ecx
    movl    %ecx, %eax
    sarl    $31, %eax
    subl    %eax, %edx
    .p2align 4
    movl    $1431655766, %eax
    leal    (%edx,%edx,2), %edx
    subl    %edx, %ecx
    movl    -32(%ebp), %edx
    addl    %ecx, %esi
    .p2align 4
    movl    (%edx,%esi,4), %ecx
.LVL25:
    imull   %ecx
    movl    %ecx, %eax
    sarl    $31, %eax
    subl    %eax, %edx
    movl    -36(%ebp), %eax
    .p2align 4
    movl    (%eax,%edx,4), %edx
    movl    %edx, -44(%ebp)
.LBE392:
.LBE391:
    movl    -40(%ebp), %edx
    movl    (%edx,%edi,4), %eax
    sall    $2, %eax
    .p2align 4
```

```
    movl    (%edx), %eax
    movl    %eax, (%esp)
    call    _Z8Compressi
.LBB397:
.LBB398:
    movl    O, %ecx
.LBE398:
.LBE397:
    movl    $1431655766, %eax
.LBB399:
.LBB400:
    movl    %ecx, -32(%ebp)
    movl    (%ecx,%esi,4), %ecx
.LVL29:
.LBE400:
.LBE399:
    imull   %ecx
    movl    %ecx, %eax
    sarl    $31, %eax
    subl    %eax, %edx
    movl    U, %eax
    movl    %eax, -36(%ebp)
    movl    (%eax,%edx,4), %eax
    testl   %eax, %eax
    jg      .L52
    movl    V, %eax
    movl    %eax, -40(%ebp)
    jmp     .L45
.LFE8:
    addl    M, %eax
    movl    (%eax), %edx
    testl   %edx, %edx
    .p2align 4
    je      .L54
    movl    -16(%ebp), %eax
    testl   %eax, %eax
    .p2align 4
    jle     .L48
    movl    -44(%ebp), %eax
    testl   %eax, %eax
    .p2align 4
    jle     .L45
    .p2align 4
.L52:
    addl    $44, %esp
    popl    %esi
    popl    %edi
.LVL26:
    popl    %ebp
    andl    $0x20ffffff, %ebp
    .p2align 4
    andl    $0x10fffff0, (%esp)
    ret
.LVL27:
    .p2align 4,,7
    .p2align 4
.L48:
    movl    -44(%ebp), %eax
    movl    -12(%ebp), %ecx
    testl   %eax, %eax
    jg      .L45
    .p2align 4
    movl    -24(%ebp), %edx
    movl    -28(%ebp), %ecx
.LVL28:
    leal    2(%edx,%edx,2), %eax
    movl    -20(%ebp), %edx
    .p2align 4
    leal    (%ecx), %ebx
    .p2align 4
    andl    $0x20ffffff, %ebx
    movl    %eax, (%ebx)
    movl    (%edx), %eax
    .p2align 4
    movl    %eax, (%esp)
    .p2align 4
    .byte 0x8d, 0xb4, 0x26, 0, 0, 0, 0
    .byte 0x8d, 0x74, 0x26, 0
    call    _Z8Compressi
.LBB397:
.LBB398:
    movl    O, %ecx
.LBE398:
```

```
.LBE397:
    movl    $1431655766, %eax
.LBB399:
.LBB400:
    movl    %ecx, -32(%ebp)
    .p2align 4
    movl    (%ecx,%esi,4), %ecx
.LVL29:
.LBE400:
.LBE399:
    imull   %ecx
    movl    %ecx, %eax
    sarl    $31, %eax
    subl    %eax, %edx
    .p2align 4
    movl    U, %eax
    movl    %eax, -36(%ebp)
    movl    (%eax,%edx,4), %eax
    testl   %eax, %eax
    .p2align 4
    jg      .L52
    movl    V, %eax
    movl    %eax, -40(%ebp)
    .p2align 4
    jmp     .L45
.LFE8:
```

Also of note is the negligible amount of code inflation between the non-sandboxed and the hybrid-sandboxed codes, readily apparent in the next listing below. Of course, it is partially offset by the presence of the bitmap of valid control transfer targets (or valid instruction boundaries) at the end of each function. As opposed to the true code inflation of padding instruction, however, the bitmap has no effect on the execution of regular code such as the functions in Listing 30. Moreover, because of their lack of indirect control transfer instructions, their control flow is sandboxed statically at load time and their unused bitmaps can safely be discarded.

It also bears repeating, that the merits of the hybrid isolation approach lie not only in its avoidance of code inflation, but in its ability to simultaneously provide the full flexibility of software fault isolation while minimizing its costs.

Listing 30: Contrast between the non-sandboxed and the hybrid-sandboxed assemblies for the `CheckHandle()` and `Compress()` routines from the Edgebreaker compression reference implementation. Note the bitmap of valid control transfer points at the end of both functions, which is computed, along with any required code instrumentation, at load time.

```
;; unsandboxed                    ;; hybrid sandboxed
```

```
;; CheckHandle()

CheckHandle:
    push   %ebp
    mov    %esp,%ebp
    sub    $0x28,%esp
    mov    %edi,-4(%ebp)
    mov    0x8(%ebp),%edi
    mov    %ebx,-12(%ebp)
    mov    %esi,-8(%ebp)
    mov    %edi,(%esp)
    call   NextEdge
    mov    O,%esi
    mov    (%esi,%eax,4),%eax
    mov    %eax,(%esp)
    call   E2T
    mov    U,%ebx
    mov    (%ebx,%eax,4),%edx
    cmp    $0x1,%edx
    jle    L1
    mov    T,%eax
    mov    %edx,0x8(%esp)
    movl   $FMT1,0x4(%esp)
    lea    0x1(%eax,%eax,2),%eax
    mov    %eax,0xc(%esp)
    mov    fhandles,%eax
    mov    %eax,(%esp)
    call   fprintf
    mov    O,%esi
    mov    U,%ebx
L1:
    mov    %edi,(%esp)
    call   PrevEdge
    mov    (%esi,%eax,4),%eax
    mov    %eax,(%esp)
    call   E2T
    mov    (%ebx,%eax,4),%edx
    cmp    $0x1,%edx
    jle    L2
    mov    T,%eax
    mov    %edx,0x8(%esp)
    movl   $FMT1,0x4(%esp)
    lea    0x2(%eax,%eax,2),%eax
    mov    %eax,0xc(%esp)
    mov    fhandles,%eax
    mov    %eax,(%esp)
    call   fprintf
L2:
    mov    -12(%ebp),%ebx
    mov    -8(%ebp),%esi
    mov    -4(%ebp),%edi
    mov    %ebp,%esp
    pop    %ebp
    ret
```

```
;; CheckHandle()

CheckHandle:
    push   %ebp
    mov    %esp,%ebp
    sub    $0x28,%esp
    mov    %edi,-4(%ebp)
    mov    0x8(%ebp),%edi
    mov    %ebx,-12(%ebp)
    mov    %esi,-8(%ebp)
    mov    %edi,(%esp)
    call   NextEdge
    mov    O,%esi
    mov    (%esi,%eax,4),%eax
    mov    %eax,(%esp)
    call   E2T
    mov    U,%ebx
    mov    (%ebx,%eax,4),%edx
    cmp    $0x1,%edx
    jle    L1
    mov    T,%eax
    mov    %edx,0x8(%esp)
    movl   $FMT1,0x4(%esp)
    lea    0x1(%eax,%eax,2),%eax
    mov    %eax,0xc(%esp)
    mov    fhandles,%eax
    mov    %eax,(%esp)
    call   fprintf
    mov    O,%esi
    mov    U,%ebx
L1:
    mov    %edi,(%esp)
    call   PrevEdge
    mov    (%esi,%eax,4),%eax
    mov    %eax,(%esp)
    call   E2T
    mov    (%ebx,%eax,4),%edx
    cmp    $0x1,%edx
    jle    L2
    mov    T,%eax
    mov    %edx,0x8(%esp)
    movl   $FMT1,0x4(%esp)
    lea    0x2(%eax,%eax,2),%eax
    mov    %eax,0xc(%esp)
    mov    fhandles,%eax
    mov    %eax,(%esp)
    call   fprintf
L2:
    mov    -12(%ebp),%ebx
    mov    -8(%ebp),%esi
    mov    -4(%ebp),%edi
    mov    %ebp,%esp
    pop    %ebp
    ud2a
```

```
    nop
    lea     0x0(%esi),%esi
```

```
    ret
    nop
    lea     0x0(%esi),%esi

CheckHandle_tgtmap:
    .byte 0x4b,0x92,0x24,0x04
    .byte 0x49,0x08,0x92,0x42
    .byte 0x04,0x44,0x84,0x84
    .byte 0x20,0x48,0x48,0x42
    .byte 0x52,0x88,0x80,0x88
    .byte 0x90,0x90,0xa4,0x19
    .byte 00
```

```
;; unsandboxed
;; Compress()

Compress:
    push    %ebp
    mov     %esp,%ebp
    push    %edi
    push    %esi
    push    %ebx
    sub     $0x1c,%esp
    mov     0x8(%ebp),%esi
    mov     U,%ebx
L1:
    mov     %esi,(%esp)
    call    E2T
    movl    $0x1,(%ebx,%eax,4)
    mov     %eax,-16(%ebp)
    addl    $0x1,T
    mov     %esi,(%esp)
    call    CheckHandle
    mov     V,%eax
    mov     (%eax,%esi,4),%edx
    mov     M,%eax
    mov     (%eax,%edx,4),%eax
    test    %eax,%eax
    je      L2
    mov     O,%ebx
    mov     %esi,(%esp)
    mov     U,%edi
    mov     %ebx,0x4(%esp)
    call    RightTri
    mov     %eax,(%esp)
    call    E2T
    mov     (%edi,%eax,4),%eax
    test    %eax,%eax
    jle     L3
    mov     %ebx,0x4(%esp)
    mov     %esi,(%esp)
    call    LeftTri
    mov     %eax,(%esp)
```

```
;; hybrid sandboxed
;; Compress()

Compress:
    push    %ebp
    mov     %esp,%ebp
    push    %edi
    push    %esi
    push    %ebx
    sub     $0x1c,%esp
    mov     0x8(%ebp),%esi
    mov     U,%ebx
L1:
    mov     %esi,(%esp)
    call    E2T
    movl    $0x1,(%ebx,%eax,4)
    mov     %eax,-16(%ebp)
    addl    $0x1,T
    mov     %esi,(%esp)
    call    CheckHandle
    mov     V,%eax
    mov     (%eax,%esi,4),%edx
    mov     M,%eax
    mov     (%eax,%edx,4),%eax
    test    %eax,%eax
    je      L2
    mov     O,%ebx
    mov     %esi,(%esp)
    mov     U,%edi
    mov     %ebx,0x4(%esp)
    call    RightTri
    mov     %eax,(%esp)
    call    E2T
    mov     (%edi,%eax,4),%eax
    test    %eax,%eax
    jle     L3
    mov     %ebx,0x4(%esp)
    mov     %esi,(%esp)
    call    LeftTri
    mov     %eax,(%esp)
```

```
        call    E2T
        mov     (%edi,%eax,4),%eax
        test    %eax,%eax
        jg      L5
        mov     fclers,%eax
        movl    $0x52,0x8(%esp)
        movl    $FMT2,0x4(%esp)
        mov     %eax,(%esp)
        call    fprintf
        mov     0,%eax
        mov     %esi,(%esp)
        mov     %eax,0x4(%esp)
        call    LeftTri
        mov     U,%ebx
        mov     %eax,%esi
        jmp     L1
L2:
        mov     fclers,%eax
        movl    $0x43,0x8(%esp)
        movl    $FMT2,0x4(%esp)
        mov     %eax,(%esp)
        call    fprintf
        mov     %esi,(%esp)
        call    EncodeDelta
        mov     V,%eax
        mov     (%eax,%esi,4),%edx
        mov     M,%eax
        movl    $0x1,(%eax,%edx,4)
        mov     0,%eax
        mov     %esi,(%esp)
        mov     %eax,0x4(%esp)
        call    RightTri
        mov     U,%ebx
        mov     %eax,%esi
        jmp     L1
L3:
        mov     %ebx,0x4(%esp)
        mov     %esi,(%esp)
        call    LeftTri
        mov     %eax,(%esp)
        call    E2T
        mov     (%edi,%eax,4),%eax
        test    %eax,%eax
        jle     L4
        mov     fclers,%eax
        movl    $0x4c,0x8(%esp)
        movl    $FMT2,0x4(%esp)
        mov     %eax,(%esp)
        call    fprintf
        mov     0,%eax
        mov     %esi,(%esp)
        mov     %eax,0x4(%esp)
        call    RightTri
        mov     U,%ebx
```

```
    mov     %eax,%esi
    jmp     L1
L4:
    mov     T,%eax
    mov     -16(%ebp),%edx
    lea     0x2(%eax,%eax,2),%eax
    mov     %eax,(%edi,%edx,4)
    mov     fclers,%eax
    movl    $0x53,0x8(%esp)
    movl    $FMT2,0x4(%esp)
    mov     %eax,(%esp)
    call    fprintf
    mov     O,%eax
    mov     %esi,(%esp)
    mov     %eax,0x4(%esp)
    call    RightTri
    mov     %eax,(%esp)
    call    Compress
    mov     O,%eax
    mov     %esi,(%esp)
    mov     %eax,0x4(%esp)
    call    LeftTri
    mov     %eax,(%esp)
    mov     %eax,%esi
    call    E2T
    mov     U,%ebx
    mov     (%ebx,%eax,4),%edi
    test    %edi,%edi
    jle     L1
    add     $0x1c,%esp
    pop     %ebx
    pop     %esi
    pop     %edi
    pop     %ebp
    ret
    mov     fclers,%eax
    movl    $0x45,0x8(%esp)
    movl    $FMT2,0x4(%esp)
    mov     %eax,(%esp)
    call    fprintf
    add     $0x1c,%esp
L5:
    pop     %ebx
    pop     %esi
    pop     %edi
    pop     %ebp
    ret
    nop
    lea     0x0(%esi),%esi
```

```
    mov     %eax,%esi
    jmp     L1
L4:
    mov     T,%eax
    mov     -16(%ebp),%edx
    lea     0x2(%eax,%eax,2),%eax
    mov     %eax,(%edi,%edx,4)
    mov     fclers,%eax
    movl    $0x53,0x8(%esp)
    movl    $FMT2,0x4(%esp)
    mov     %eax,(%esp)
    call    fprintf
    mov     O,%eax
    mov     %esi,(%esp)
    mov     %eax,0x4(%esp)
    call    RightTri
    mov     %eax,(%esp)
    call    Compress
    mov     O,%eax
    mov     %esi,(%esp)
    mov     %eax,0x4(%esp)
    call    LeftTri
    mov     %eax,(%esp)
    mov     %eax,%esi
    call    E2T
    mov     U,%ebx
    mov     (%ebx,%eax,4),%edi
    test    %edi,%edi
    jle     L1
    add     $0x1c,%esp
    pop     %ebx
    pop     %esi
    pop     %edi
    pop     %ebp
    ud2a
    ret
    mov     fclers,%eax
    movl    $0x45,0x8(%esp)
    movl    $FMT2,0x4(%esp)
    mov     %eax,(%esp)
    call    fprintf
    add     $0x1c,%esp
L5:
    pop     %ebx
    pop     %esi
    pop     %edi
    pop     %ebp
    ud2a
    ret
    nop
    lea     0x0(%esi),%esi

Compress_tgtmap:
    .byte   0x7b,0x12,0x24,0x04
```

```
.byte 0x12,0x48,0x08,0x09
.byte 0x29,0x08,0x12,0x44
.byte 0x48,0x48,0x41,0x24
.byte 0x24,0xa4,0x20,0x04
.byte 0x04,0x24,0x84,0x44
.byte 0x08,0x0a,0x21,0x20
.byte 0x20,0x21,0x21,0x24
.byte 0x04,0x42,0x22,0x04
.byte 0x85,0x48,0x48,0x48
.byte 0x85,0x80,0x80,0x84
.byte 0x90,0x08,0x41,0x21
.byte 0x24,0x12,0x02,0x02
.byte 0x12,0x42,0x22,0x24
.byte 0x84,0x44,0x48,0x21
.byte 0x48,0x41,0x3e,0x21
.byte 0x20,0x20,0x21,0x9f
.byte 0x01
```

B.2 GrayEdge

GrayEdge, our second example, is an image transcoding kernel for the CameraCast extensible remote video sensor system [52]. It consists of two consecutive steps: (a) image gray-scaling, and (b) edge detection. As with the EdgeBreaker example, the listings for the full kernel would be too unwieldy to include in their entirety, so we will focus on the relatively shorter gray-scaling part.

The logic and points of our argument are shared with the ones already presented in the previous section, so we will not repeat them, but rather focus on the important differences.

Unlike EdgeBreaker, the structure of the image transcoding kernel is more monolithic, consisting of only two but larger functions, and exhibiting a less conditional and more cyclic structure. This is typical in graphics codes where transformations often process the image one pixel at a time and iterate over the entire frame.

The outcome is a relatively simple assembly that, which unfortunately, is heavily memory bound and results in a large number of cache read and/or write misses as evidenced in Table 7. That is largely due to the size of its input and output data (the video frames) and despite the good spatial locality of the sequence of accesses.

From an isolation technique's point of view, however, the GrayEdge code, shown in Listing 31, plays directly to the strengths of the hybrid approach. Because of its looping

Table 7: Cache profile generated with Cachegrind for the gray-scaling and edge detection image processing kernel operating on the image from Figure 31.

Ir	Dr	D1mr	D2mr	Dw	D1mw	D2mw	Function
9,527,536	4,917,605	28,800	28,800	307,685	9,600	9,600	grayImage()
39,085,773	20,643,798	9,560	9,560	2,032,467	9,581	9,581	edgeImage()

structure, it exhibits a large number of indirect memory reference instruction (by the number of their execution) and almost no indirect control transfer instructions, save for the two function returns, and even those are only executed but once each. Thus, GrayEdge comes close to being the perfect candidate for hybrid code isolation and achieves an almost native execution speed as already demonstrated in Chapter 4.

Listing 31: Integer arithmetic C implementation of the gray-scaling routine from the GrayEdge image transcoding kernel.

```
int grayImage(unsigned char * in, unsigned char * out, int x, int y)
{
  int col, row, indx, outIndx;

  indx = 0;
  outIndx = 0;
  for (row = 0; row < y; ++row) {
    for ( col = 0; col < x; ++col) {
      out[outIndx] =
        (unsigned char) (1 * (in[indx]   >> 2) +
                         5 * (in[indx+1] >> 3) +
                         1 * (in[indx+2] >> 3));
      indx += 3;
      ++outIndx;
    }
  }
  return 0;
}
```

In the remainder of this section, we provide comparison assembly listings for the non-sandboxed *vs.* PittSFIeld-sandboxed and non-sandboxed *vs.* hybrid-sandboxed gray-scaling routine shown above.

Listing 32: Contrast between the non-sandboxed and the PittSFIeld-sandboxed assemblies for the `grayImage()` routine from the GrayEdge image processing benchmark.

```
;; unsandboxed
;; grayImage()

.globl grayImage
grayImage:
```

```
;; PittSFIeld sandboxed
;; grayImage()

    .p2align 4,,15
.globl grayImage
```

```
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        subl    $16, %esp
        movl    $0, -12(%ebp)
        movl    $0, -8(%ebp)
        movl    $0, -16(%ebp)
        jmp     .L2
.L3:
        movl    $0, -20(%ebp)
        jmp     .L4
.L5:
        movl    -8(%ebp), %eax
        movl    %eax, %ecx
        addl    12(%ebp), %ecx
        movl    -12(%ebp), %eax
        addl    8(%ebp), %eax
        movzbl  (%eax), %eax
        movl    %eax, %ebx
        shrb    $2, %bl
        movl    -12(%ebp), %eax
        addl    8(%ebp), %eax
        addl    $1, %eax
        movzbl  (%eax), %eax
        shrb    $3, %al
        movzbl  %al, %edx
        movl    %edx, %eax
        sall    $2, %eax
        addl    %edx, %eax
        leal    (%ebx,%eax), %edx
        movl    -12(%ebp), %eax
        addl    8(%ebp), %eax
        addl    $2, %eax
        movzbl  (%eax), %eax
        shrb    $3, %al
        leal    (%edx,%eax), %eax
        movb    %al, (%ecx)
        addl    $3, -12(%ebp)
        addl    $1, -8(%ebp)
        addl    $1, -20(%ebp)
.L4:
        movl    -20(%ebp), %eax
        cmpl    16(%ebp), %eax
        jl      .L5
        addl    $1, -16(%ebp)
.L2:
        movl    -16(%ebp), %eax
        cmpl    20(%ebp), %eax
        jl      .L3
        movl    $0, %eax
        addl    $16, %esp
        popl    %ebx
        popl    %ebp
        ret
        .p2align 4
grayImage:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %edi
        pushl   %esi
        subl    $12, %esp
        movl    20(%ebp), %edx
        .p2align 4
        andl    $0x20ffffff, %esp
        testl   %edx, %edx
        jle     .L2f
        .p2align 4
        movl    $0, -16(%ebp)
        movl    $0, -12(%ebp)
        .p2align 4
        movl    $0, -20(%ebp)
        .p2align 4,,7
        .p2align 4
.L4:
        movl    16(%ebp), %eax
        testl   %eax, %eax
        jle     .L7f
        movl    -16(%ebp), %ecx
        xorl    %esi, %esi
        .p2align 4
        movl    -12(%ebp), %edi
        addl    8(%ebp), %ecx
        addl    12(%ebp), %edi
        .p2align 4,,7
        .p2align 4
.L5:
        movzbl  2(%ecx), %edx
        addl    $1, %esi
        movzbl  (%ecx), %eax
        .p2align 4
        shrb    $3, %dl
        shrb    $2, %al
        .p2align 4
        addl    %eax, %edx
        movzbl  1(%ecx), %eax
        addl    $3, %ecx
        .p2align 4
        shrb    $3, %al
        movzbl  %al, %eax
        leal    (%eax,%eax,4), %eax
        addl    %eax, %edx
        .p2align 4
        leal    (%edi), %ebx
        .p2align 4
        andl    $0x20ffffff, %ebx
        movb    %dl, (%ebx)
        addl    $1, %edi
        .p2align 4
```

```
        cmpl    16(%ebp), %esi
        jne     .L5f
        leal    (%esi,%esi,2), %eax
        addl    %esi, -12(%ebp)
        .p2align 4
        addl    %eax, -16(%ebp)
        .p2align 4
.L7:
        addl    $1, -20(%ebp)
        movl    20(%ebp), %eax
        cmpl    %eax, -20(%ebp)
        .p2align 4
        jne     .L4f
        .p2align 4
.L2:
        addl    $12, %esp
        xorl    %eax, %eax
        popl    %esi
        popl    %edi
        popl    %ebp
        andl    $0x20ffffff, %ebp
        .p2align 4
        andl    $0x10fffff0, (%esp)
        ret
```

Listing 33: Contrast between the non-sandboxed and the hybrid-sandboxed assemblies for the `grayImage()` routine from the GrayEdge image processing benchmark.

```
;; unsandboxed
;; grayImage()

        .p2align 4,,15
.globl grayImage
        .p2align 4
grayImage:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %edi
        pushl   %esi
        subl    $12, %esp
        movl    20(%ebp), %edx
        .p2align 4
        andl    $0x20ffffff, %esp
        testl   %edx, %edx
        jle     .L2f
        .p2align 4
        movl    $0, -16(%ebp)
        movl    $0, -12(%ebp)
        .p2align 4
        movl    $0, -20(%ebp)
        .p2align 4,,7
        .p2align 4
.L4:
        movl    16(%ebp), %eax
```

```
;; hybrid sandboxed
;; grayImage()

        .p2align 4,,15
.globl grayImage
        .p2align 4
grayImage:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %edi
        pushl   %esi
        subl    $12, %esp
        movl    20(%ebp), %edx
        .p2align 4
        andl    $0x20ffffff, %esp
        testl   %edx, %edx
        jle     .L2f
        .p2align 4
        movl    $0, -16(%ebp)
        movl    $0, -12(%ebp)
        .p2align 4
        movl    $0, -20(%ebp)
        .p2align 4,,7
        .p2align 4
.L4:
        movl    16(%ebp), %eax
```

```
        testl   %eax, %eax
        jle     .L7f
        movl    -16(%ebp), %ecx
        xorl    %esi, %esi
        .p2align 4
        movl    -12(%ebp), %edi
        addl    8(%ebp), %ecx
        addl    12(%ebp), %edi
        .p2align 4,,7
        .p2align 4
.L5:
        movzbl  2(%ecx), %edx
        addl    $1, %esi
        movzbl  (%ecx), %eax
        .p2align 4
        shrb    $3, %dl
        shrb    $2, %al
        .p2align 4
        addl    %eax, %edx
        movzbl  1(%ecx), %eax
        addl    $3, %ecx
        .p2align 4
        shrb    $3, %al
        movzbl  %al, %eax
        leal    (%eax,%eax,4), %eax
        addl    %eax, %edx
        .p2align 4
        leal    (%edi), %ebx
        .p2align 4
        andl    $0x20ffffff, %ebx
        movb    %dl, (%ebx)
        addl    $1, %edi
        .p2align 4
        cmpl    16(%ebp), %esi
        jne     .L5f
        leal    (%esi,%esi,2), %eax
        addl    %esi, -12(%ebp)
        .p2align 4
        addl    %eax, -16(%ebp)
        .p2align 4
.L7:
        addl    $1, -20(%ebp)
        movl    20(%ebp), %eax
        cmpl    %eax, -20(%ebp)
        .p2align 4
        jne     .L4f
        .p2align 4
.L2:
        addl    $12, %esp
        xorl    %eax, %eax
        popl    %esi
        popl    %edi
        popl    %ebp
        andl    $0x20ffffff, %ebp
```

```
        testl   %eax, %eax
        jle     .L7f
        movl    -16(%ebp), %ecx
        xorl    %esi, %esi
        .p2align 4
        movl    -12(%ebp), %edi
        addl    8(%ebp), %ecx
        addl    12(%ebp), %edi
        .p2align 4,,7
        .p2align 4
.L5:
        movzbl  2(%ecx), %edx
        addl    $1, %esi
        movzbl  (%ecx), %eax
        .p2align 4
        shrb    $3, %dl
        shrb    $2, %al
        .p2align 4
        addl    %eax, %edx
        movzbl  1(%ecx), %eax
        addl    $3, %ecx
        .p2align 4
        shrb    $3, %al
        movzbl  %al, %eax
        leal    (%eax,%eax,4), %eax
        addl    %eax, %edx
        .p2align 4
        leal    (%edi), %ebx
        .p2align 4
        andl    $0x20ffffff, %ebx
        movb    %dl, (%ebx)
        addl    $1, %edi
        .p2align 4
        cmpl    16(%ebp), %esi
        jne     .L5f
        leal    (%esi,%esi,2), %eax
        addl    %esi, -12(%ebp)
        .p2align 4
        addl    %eax, -16(%ebp)
        .p2align 4
.L7:
        addl    $1, -20(%ebp)
        movl    20(%ebp), %eax
        cmpl    %eax, -20(%ebp)
        .p2align 4
        jne     .L4f
        .p2align 4
.L2:
        addl    $12, %esp
        xorl    %eax, %eax
        popl    %esi
        popl    %edi
        popl    %ebp
        andl    $0x20ffffff, %ebp
```

```
.p2align 4
andl    $0x10ffffff0, (%esp)
ret
```

```
.p2align 4
andl    $0x10ffffff0, (%esp)
ud2a
ret
```

REFERENCES

[1] 4Front Technologies / XMMS Team, *X MultiMedia System*. `http://www.xmms.org`. Accessed April, 2006.

[2] Accetta, M. J., Baron, R. V., Bolosky, W. J., Golub, D. B., Rashid, R. F., Tevananian, A., and Young, M., "Mach: A new kernel foundation for Unix development," in *Proceedings of the Summer 1986 USENIX Conference*, pp. 99–112, 1986.

[3] Adobe Systems Inc. formerly Macromedia, *Macromedia ShockWave player*. `http://www.macromedia.com/shockwave/`. Accessed April 2006.

[4] Aiken, M., Fahndrich, M., Hunt, C. H. G. C., and Larus, J. R., "Deconstructing process isolation," Tech. Rep. MSR-TR-2006-43, Microsoft Corporation, Redmond, WA, April 2006.

[5] Anagnostakis, K. G., Greenwald, M., Ioannidis, S., and Miltchev, S., "Open packet monitoring on FLAME: Safety, performance and applications," in *Proceedings of the* 4^{th} *International Working Conference on Active Networks (IWAN'02)*, December 2002.

[6] Anagnostakis, K. G., Ioannidis, S., Miltchev, S., Ioannidis, J., Greenwald, M., and Smith, J. M., "Efficient packet monitoring for network management," in *Proceedings of the* 8^{th} *IEEE Network Operations and Management Symposium (NOMS'02)*, IEEE, April 2002.

[7] Ball, T., Cook, B., Levin, V., and Rajamani, S. K., "SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft," Tech. Rep. MSR-TR-2004-8, Microsoft Corp., Redmont, WA, USA, January 2004.

[8] Ball, T. and Rajamani, S. K., "Automatically validating temporal safety properties of interfaces," in *Proceedings of the* 8^{th} *International SPIN Workshop on Model Checking of Software (SPIN'2001)*, May 2001.

[9] Ball, T. and Rajamani, S. K., "The SLAM project: Debugging system software via static analysis," in *Proceedings of the* 29^{th} *Annual Symposium on Principles of Programming Languages*, pp. 1–3, January 2002.

[10] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A., "Xen and the art of virtualization," in *Proceedings of the* 19^{th} *Symposium on Operating Systems Principles*, ACM Press, October 2003.

[11] Bennett, S. M., Neiger, G., Cota-Robles, E. C., Jeyasingh, S., Kagi, A., Kozuch, M. A., Uhlig, R. A., Smith, L., Rodgers, D., Glew, A., and Boleyn, E., "Methods and systems to control virtual machines." Unite States Patent Application 2,004,0117539, June 2004.

[12] Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Chambers, C., and Eggers, S., "Extensibility, safety and performance in the SPIN operating system," in *Proceedings of the* 15^{th} *ACM Symposium on Operating Systems Principles*, pp. 267–283, ACM Press, 1995.

[13] Bos, H. and Samwel, B., "Safe kernel programming in the OKE," in *Proceedings of the* 5^{th} *International Conference on Open Architectures and Network Programming*, pp. 141–152, IEEE, 2002.

[14] Bos, H. and Samwel, B., "HOKES/POKES: Lightweight resource sharing," in *Proceedings of the* 3^{rd} *International Conference on Embedded Software (EMSOFT'03)*, ACM Press, October 2003.

[15] Bos, H., Samwel, B., Cristea, M., and Anagnostakis, K. G., "Safe execution of untrusted applications on embedded network processors," in *Domain-Specific Processors: Systems, Architectures, Modeling, and Simulation* (Bhattacharyya, S. S., ed.), Marcel Dekker, Inc., 2003.

[16] Chiueh, T., Venkitachalam, G., and Pradhan, P., "Integrating segmentation and paging protection for safe, efficient and transparent software extensions," in *Proceedings of the* 17^{th} *ACM Symposium on Operating Systems Principles*, pp. 140–153, ACM Press, 1999.

[17] Condit, J., Harren, M., McPeak, S., Necula, G. C., and Weimer, W., "CCured in the real world," in *Proceedings of the ACM SIGPLAN Conference on Programming language Design and Implementation (PLDI '03)*, (New York, NY, USA), pp. 232–244, ACM Press, 2003.

[18] Connectix, Corp., "The technology of Virtual PC," 2000.

[19] Corbet, J., Rubini, A., and Kroah-Hartman, G., *Linux Device Drivers*. O'Reilly Media, Inc., 3^{rd} ed., February 2005.

[20] Corp., M., "Guidelines for providing multimedia timer support." Microsoft Corp. web pages. `http://www.microsoft.com/whdc/system/CEC/mm-timer.mspx`. Accessed October, 2006.

[21] Crary, K. and Morrisett, J. G., "Type structure for low-level programming languages," in *Proceedings of the* 26^{th} *International Colloquium on Automata, Languages and Programming (ICAL '99)*, (London, UK), pp. 40–54, Springer-Verlag, 1999.

[22] Engler, D. R. and Kaashoek, M. F., "Exterminate all operating system abstractions," in *Proceedings of the* 5^{th} *Workshop on Hot Topics in Operating Systems (HotOS-V)*, (Orcas Island, WA), pp. 78–83, IEEE Computer Society, May 1995.

[23] Eric Hamiter, *BugMeNot extension plugin for FireFox*. `http://roachfiend.com/archives/2005/02/07/bugmenot/`. Accessed April, 2006.

[24] Ford, B., "The VX32 virtual extension environment." MIT Computer Science and Artificial Intelligence Laboratory web pages. `http://pdos.csail.mit.edu/~baford/vm/`. Accessed April, 2006.

[25] Ford, B., "VXA: A virtual architecture for durable compressed archives," in *Proceedings of the 4^{th} USENIX Conference on File and Storage Technologies (FAST '05)*, December 2005.

[26] The Free Software Foundation, *GDB: The GNU Project Debugger*. `http://www.gnu.org/software/gdb/gdb.html`. Accessed September, 2006.

[27] The Free Software Foundation, *The GNU Compiler Collection*. `http://gcc.gnu.org`. Accessed July, 2006.

[28] Free Software Foundation, Inc., *GNU Emacs*. `http://www.gnu.org/software/emacs/emacs.html`. Accessed April, 2006.

[29] Ganev, I., Eisenhauer, G., and Schwan, K., "Kernel Plugins: When a VM is too much," in *Proceedings of the 3^{rd} Virtual Machine Research and Technology Symposium (VM'04)*, May 2004.

[30] Ghormley, D. P., Petrou, D., Rodrigues, S. H., and Anderson, T. E., "SLIC: An extensibility system for commodity operating systems," in *Proceedings of the 1998 USENIX Annual Technical Conference*, pp. 39–52, June 1998.

[31] The GIMP Team, *The GNU Image Manipulation Program*. `http://www.gimp.org`. Accessed April, 2006.

[32] Golub, D. B., Dean, R., Forin, A., and Rashid, R. F., "Unix as an application program," in *Proceedings of the USENIX Summer Conference*, pp. 87–96, USENIX Association, June 1990.

[33] Grossman, D., "Type-safe multithreading in Cyclone," in *Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003)*, pp. 13–25, January 2003.

[34] Grossman, D., Morrisett, G., Jim, T., Hicks, M., Wang, Y., and Cheney, J., "Region-based memory management in Cyclone," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI'02)*, pp. 282–293, ACM Press, June 2002. Extended version available as Cornell CS Technical Report TR2001-1856.

[35] Guillemont, M., "The Chorus distributed operating system," in *Proceedings of the ACM International Symposium on Local Computer Networks*, pp. 207–223, ACM Press, April 1982.

[36] Halfhill, T. R., "Beyond Pentium II." BYTE.com cover story for December 1997. `http://www.byte.com/art/9712/sec5/art1.htm`. Accessed August, 2006.

[37] Harren, M. and Necula, G. C., "Lightweight wrappers for interfacing with binary code in CCured," in *Proceedings of the 3^{rd} International Symposium on Software Security (ISSS'03)*, November 2003.

[38] Härtig, H., Hohmuth, M., Liedtke, J., Schönberg, S., and Wolter, J., "The performance of μ-kernel-based systems," in *Proceedings of the 16^{th} ACM Symposium on Operating Systems Principles*, pp. 66–77, ACM Press, 1997.

[39] Hewlett Packard Corporation, Palo Alto, CA, *PA-RISC 2.0 Architecture Reference Manual*, 1994.

[40] Hicks, M., Weirich, S., and Crary, K., "Safe and flexible dynamic linking of native code," in *Proceedings of the* 3^{rd} *ACM SIGPLAN Workshop on Types in Compilation (TIC 2000)*, September 2000.

[41] Hicks, M., Morrisett, G., Grossman, D., and Jim, T., "Safe and flexible memory management in Cyclone," Tech. Rep. Technical Report CS-TR-4514, University of Maryland, July 2003.

[42] Hicks, M., Morrisett, G., Grossman, D., and Jim, T., "Experience with safe manual memory-management in Cyclone," in *Proceedings of the International Symposium on Memory Management (ISMM'04)*, ACM Press, October 2004.

[43] Hsieh, W., Fiuczynski, M., Garrett, C., Savage, S., Becker, D., and Bershad, B., "Language support for extensible operating systems," in *Proceedings of the Workshop on Compiler Support for System Software*, February 1996.

[44] Hunt, G. C., Larus, J. R., Abadi, M., Aiken, M., Barham, P., Fahndrich, M., Hodson, C. H. O., Levi, S., Murphy, N., Steensgaard, B., Tarditi, D., Wobber, T., and Zill, B., "An overview of the singularity project," Tech. Rep. MSR-TR-2005-135, Microsoft Corporation, Redmond, WA, October 2005.

[45] Intel Corporation, Santa Clara, CA, *IA-PC HPET Specification, Rev. 1.0a*, October 2004.

[46] Intel Corporation, Santa Clara, CA, *IA-32 Intel ® Architecture Software Developer's Manual, Volume 1: Basic Architecture*, June 2005.

[47] Intel Corporation, Santa Clara, CA, *IA-32 Intel ® Architecture Software Developer's Manual, Volume 2A: Instruction Set Reference, A-M*, June 2005.

[48] Intel Corporation, Santa Clara, CA, *IA-32 Intel ® Architecture Software Developer's Manual, Volume 2B: Instruction Set Reference, N-Z*, June 2005.

[49] Intel Corporation, Santa Clara, CA, *IA-32 Intel ® Architecture Software Developer's Manual, Volume 3: Architecture and Programming Manual*, June 2005.

[50] Ioannidis, S., Anagnostakis, K. G., Ioannidis, J., and Keromytis, A., "xPF: Packet filtering for low-cost network monitoring," in *Proceedings of the IEEE Workshop on High Performance Switching and Routing (HPSR'02)*, IEEE, May 2002.

[51] Jim, T., Morrisett, G., Grossman, D., Hicks, M., Cheney, J., and Wang, Y., "Cyclone: A safe dialect of C," in *Proceedings of the USENIX 2002 Annual Technical Conference*, June 2002.

[52] Kong, J., Ganev, I., Schwan, K., and Widener, P., "Cameracast: Flexible access to remote video sensors," in *MMCN '07: Proceedings of the* 14^{th} *Annual Multimedia Computing and Networking Conference (MMCN'07)*, (New York, NY, USA), ACM Press, 2007.

[53] Liedtke, J., "On μ-kernel construction," in *Proceedings of the* 15^{th} *ACM Symposium on Operating Systems Principles*, pp. 237–250, ACM Press, 1995.

[54] Liedtke, J., "A persistent system in real use – experiences of the first 13 years," in *Proceedings of the* 3^{rd} *International Workshop on Object Orientation in Operating Systems (IWOOS)*, pp. 2–11, IEEE Computer Society Press, December 1993.

[55] Liedtke, J., "Improved address-space switching on Pentium processors by transparently multiplexing user address spaces," Tech. Rep. 933, National Research Center for Information Technology, November 1995.

[56] Liedtke, J., "Toward real microkernels," *Communication of the ACM*, vol. 39, no. 9, pp. 70–77, 1996.

[57] Love, R., *Linux Kernel Development*. Novell Press, 2^{nd} ed., January 2005.

[58] Mauro, J. and McDougall, R., *Solaris Internals: Core Kernel Architecture*. Sun Microsystems Press, 1^{st} ed., October 2000.

[59] McCamant, S., "PittSFIeld." MIT Computer Science and Artificial Intelligence Laboratory web pages. `http://pag.csail.mit.edu/~smcc/projects/pittsfield/`. Accessed April, 2006.

[60] McCamant, S. and Morrisett, G., "Efficient, verifiable binary sandboxing for a CISC architecture," Tech. Rep. MIT-LCS-TR-988, MIT Laboratory for Computer Science, May 2005. Also available as MIT Computer Science and Artificial Intelligence Laboratory Technical Report 2005-030 (MIT-CSAIL-TR-2005-030).

[61] McCanne, S. and Jacobson, V., "The BSD packet filter: A new architecture for user-level packet capture," in *Proceedings of the Winter 1993 USENIX Conference*, pp. 259–269, 1993.

[62] Microsoft Corp., "Microsoft virtual pc 2004." `http://www.microsoft.com/windows/virtualpc/`. Accessed Aug, 2006., 2000.

[63] Microsoft Corp., "Device-driver performance considerations for multimedia platforms." `http://www.microsoft.com/whdc/driver/perform/mmdrv.mspx`. Accessed Oct, 2006., October 2006.

[64] MIPS Technologies, Inc., Mountain View, CA, *MIPS32 ® Architecture for Programmers, Volume II: The MIPS32 ® Instruction Set*, July 2005.

[65] MIPS Technologies, Inc., Mountain View, CA, *MIPS64 ® Architecture for Programmers, Volume II: The MIPS64 ® Instruction Set*, July 2005.

[66] Moore, B., Slabach, T., and Schaelicke, L., "Profiling interrupt handler performance through kernel instrumentation," in *Proceedings of the* 21^{st} *IEEE International Conference on Computer Design (ICCD'03)*, IEEE, October 2003.

[67] Moore, E. F., "A simplified universal Turing machine," in *ACM '52: Proceedings of the 1952 ACM national meeting (Toronto)*, (New York, NY, USA), pp. 50–54, ACM Press, 1952.

[68] Morris, R., Kohler, E., Jannotti, J., and Kaashoek, M. F., "The Click modular router," in *Proceedings of the* 17^{th} *ACM Symposium on Operating Systems Principles (SOSP'99)*, pp. 217–231, December 1999.

[69] Morrisett, G., Crary, K., Glew, N., Grossman, D., Samuels, R., Smith, F., Walker, D., Weirich, S., and Zdancewic, S., "TALx86: A realistic typed assembly language," in *Proceedings of the* 2^{nd} *ACM SIGPLAN Workshop on Computer Support for System Software*, (Atlanta, GA), pp. 25–35, 1999. Published as INRIA Technical Report 0288, March, 1999.

[70] Morrisett, G., Walker, D., Crary, K., and Glew, N., "From System F to typed assembly language," *ACM Transactions on Programming Languages and Systems*, vol. 21, no. 3, pp. 527–568, 1999.

[71] Mozilla Corporation, *FireFox web broser*. `http://www.mozilla.org/products/firefox/`. Accessed April, 2006.

[72] The MPlayer Project, *MPlayer movie player*. `http://mplayerhq.hu/homepage/design7/news.html`. Accessed April, 2006.

[73] Mullender, S. J., van Rossum, G., Tanenbaum, A. S., van Renesse, R., and van Staveren, H., "Amoeba: A distributed operating system for the 1990s," *Computer*, vol. 23, pp. 44–53, May 1990.

[74] Necula, G. C., "Proof-carrying code," in *Proceedings of the* 24^{th} *Annual Symposium on Principles of Programming Languages*, pp. 106–119, ACM Press, 1997.

[75] Necula, G. C., Condit, J., Harren, M., McPeak, S., and Weimer, W., "CCured: Type-safe retrofitting of legacy software," in *Proceedings of the* 29^{th} *Annual ACM Symposium on Principles of Programming Languages (PoPL'02)*, pp. 128–139, ACM Press, January 2002.

[76] Necula, G. C., Condit, J., Harren, M., McPeak, S., and Weimer, W., "CCured: Type-safe retrofitting of legacy software," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 3, pp. 477–526, 2005.

[77] Necula, G. C. and Lee, P., "Safe kernel extensions without runtime checking," in *Proceedings of the* 2^{nd} *Symposium on Operating Systems Design and Implementation*, USENIX, 1996.

[78] Nethercote, N. and Mycroft, A., "The cache behaviour of large lazy functional programs on stock hardware," in *MSP '02: Proceedings of the 2002 workshop on Memory system performance*, (New York, NY, USA), pp. 44–55, ACM Press, 2002.

[79] Nethercote, N. and Seward, J., "Cachegrind: a cache profiler." Part of the Valgrind simulation-based debugging and profiling tool suite. `http://valgrind.org/docs/manual/cg-manual.html`. Accessed August, 2006.

[80] Oney, W., *Programming the Microsoft Windows Driver Model*. Microsoft Press, 2^{nd} ed., December 2002.

[81] Oppliger, R., *Internet and Intranet Security*. Artech House Publishers, 2^{nd} ed., 2002.

[82] Pardyak, P. and Bershad, B., "Dynamic binding for an extensible system," in *Proceedings of the* 2^{nd} *USENIX Symposium on Operating Systems Design and Implementation (OSDI'96)*, pp. 201–212, October 1996.

[83] Patel, P. and Lepreau, J., "Hybrid resource control of active extensions," in *Proceedings of the* 6^{th} *IEEE Conference on Open Architectures and Network Programming (OPENARCH'03)*, IEEE, April 2003.

[84] Patel, P., Wetherall, D., Lepreau, J., and Whitaker, A., "TCP meets mobile code," in *Proceedings of the* 9^{th} *Workshop on Hot Topics in Operating Systems*, May 2003.

[85] Patel, P., Whitaker, A., Wetherall, D., Lepreau, J., and Stack, T., "Upgrading transport protocols using untrusted mobile code," in *Proceedings of the* 19^{th} *ACM Symposium on Operating System Principles*, Oct 2003.

[86] Patel, P. K., "Hybrid resource control for fast-path active extensions," Master's thesis, School of Computing, The University of Utah, December 2003.

[87] Rossignac, J., "Edgebreaker 3D Compression." Georgia Institute of Technology GVU Center web pages. `http://www.gvu.gatech.edu/~jarek/edgebreaker/`. Accessed August, 2006.

[88] Rossignac, J., "Edgebreaker: Connectivity compression for triangle meshes," *IEEE Transactions on Visualization and Computer Graphics*, vol. 5, pp. 47–61, January–March 1999.

[89] Rossignac, J., Safonova, A., and Szymczak, A., "3D compression made simple: Edgebreaker on a Corner Table," in *Proceedings of the Shape Modeling International Conference*, May 2001.

[90] Russinovich, M. E. and Solomon, D. A., *Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server(TM) 2003, Windows XP, and Windows 2000 (Pro-Developer)*. Microsoft Press, 4^{th} ed., December 2004.

[91] Seltzer, M. I., Endo, Y., Small, C., and Smith, K. A., "Dealing with disaster: Surviving misbehaved kernel extensions," in *Proceedings of the* 2^{nd} *USENIX Symposium on Operating Systems Design and Implementation*, pp. 213–227, ACM Press, 1996.

[92] Sirer, E. G., Savage, S., Pardyak, P., DeFouw, G., Alapat, M. A., and Bershad, B., "Writing an operating system using Modula-3," in *Proceedings of the Workshop on Compiler Support for System Software*, February 1996.

[93] Small, C., "MiSFIT: A tool for constructing safe extensible C++ systems," in *Proceedings of the* 3^{rd} *USENIX Conference on Object-Oriented Technologies (COOTS'97)*, June 1997.

[94] Small, C. and Seltzer, M. I., "MiSFIT: Constructing safe extensible systems," *IEEE Concurrency*, vol. 6, pp. 34–41, July–September 1998.

[95] Stallings, W., *Operating Systems*. Prentice Hall, 4^{th} ed., 2001.

[96] Swift, M. M., Bershad, B. N., and Levy, H. M., "Improving the reliability of commodity operating systems," in *Proceedings of the* 19^{th} *ACM Symposium on Operating System Principles*, Oct 2003.

[97] Tanenbaum, A. S. and Mullender, S., "An overview of the Amoeba distributed operating system," *Operating Systems Review*, vol. 15, pp. 51–64, July 1981.

[98] Turing, A. M., "On computable numbers, with an application to the Entscheidungs problem," *Proceedings of the Londong Mathematics Society*, vol. 24, no. 2, pp. 230–265, 1936.

[99] Uhlig, V., Dannowski, U., Skoglund, E., Haeberlen, A., and Heiser, G., "Performance of address-space multiplexing on the Pentium," Interner Bericht 2002-01, Fakultät für Informatik, Universität Karlsruhe, 2002.

[100] Venkitachalam, G. R., "Palladium: A system for supporting safe user extensions using segmentation hardware," Master's thesis, State University of New York at Stony Brook, May 1999.

[101] VMware, Inc., "VMware virtual platform." Technical white paper, 1999.

[102] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L., "Efficient software-based fault isolation," in *Proceedings of the* 14^{th} *ACM Symposium on Operating Systems Principles*, pp. 203–216, ACM Press, 1993.

[103] Wallach, D. A., Engler, D. R., and Kaashoek, M. F., "ASHs: Application-specific handlers for high-performance messaging," *IEEE/ACM Transactions on Networking (TON)*, vol. 5, no. 4, pp. 460–474, 1997.

[104] Wang, Y.-M., "STRIDER: A new approach to configuration and security management." Research presentation at the Georgia Institute of Technology's Center for Experimental Research in Computer Science (CERCS), November 2004.

[105] Wetzel, J., Silha, E., May, C., and Frey, B., *PowerPC Architecture Book: Book II: PowerPC Virtual Environment Architecture,* Version 2.01. International Business Machines Corporation, New York, NY, December 2003.

[106] Wetzel, J., Silha, E., May, C., and Frey, B., *PowerPC Architecture Book: Book III: PowerPC Operating Environment Architecture,* Version 2.01. International Business Machines Corporation, New York, NY, December 2003.

[107] Whitaker, A., Shaw, M., and Gribble, S. D., "Denali: A scalable isolation kernel," in *Proceedings of the Tenth ACM SIGOPS European Workshop*, September 2002.

[108] The xine-Project, *Xine multimedia player*. `http://xinehq.de`. Accessed April, 2006.

VITA

Ivan Ganev was born in Ruse, Bulgaria on April 19, 1975. Ivan obtained his B.A. degree in Computer Science from the American University in Bulgaria (AUBG) in Blagoevgrad, Bulgaria, and his M.S. and Ph.D. degrees from the Georgia Institute of Technology in Atlanta, GA.

Ivan is primarily interested in all aspects of operating systems, though his wider interests include computer architectures, virtualization, compilers, and computer networks.